

# microbee<sup>®</sup>

Australia's home grown  
computer



Microbee

\$9.95\*

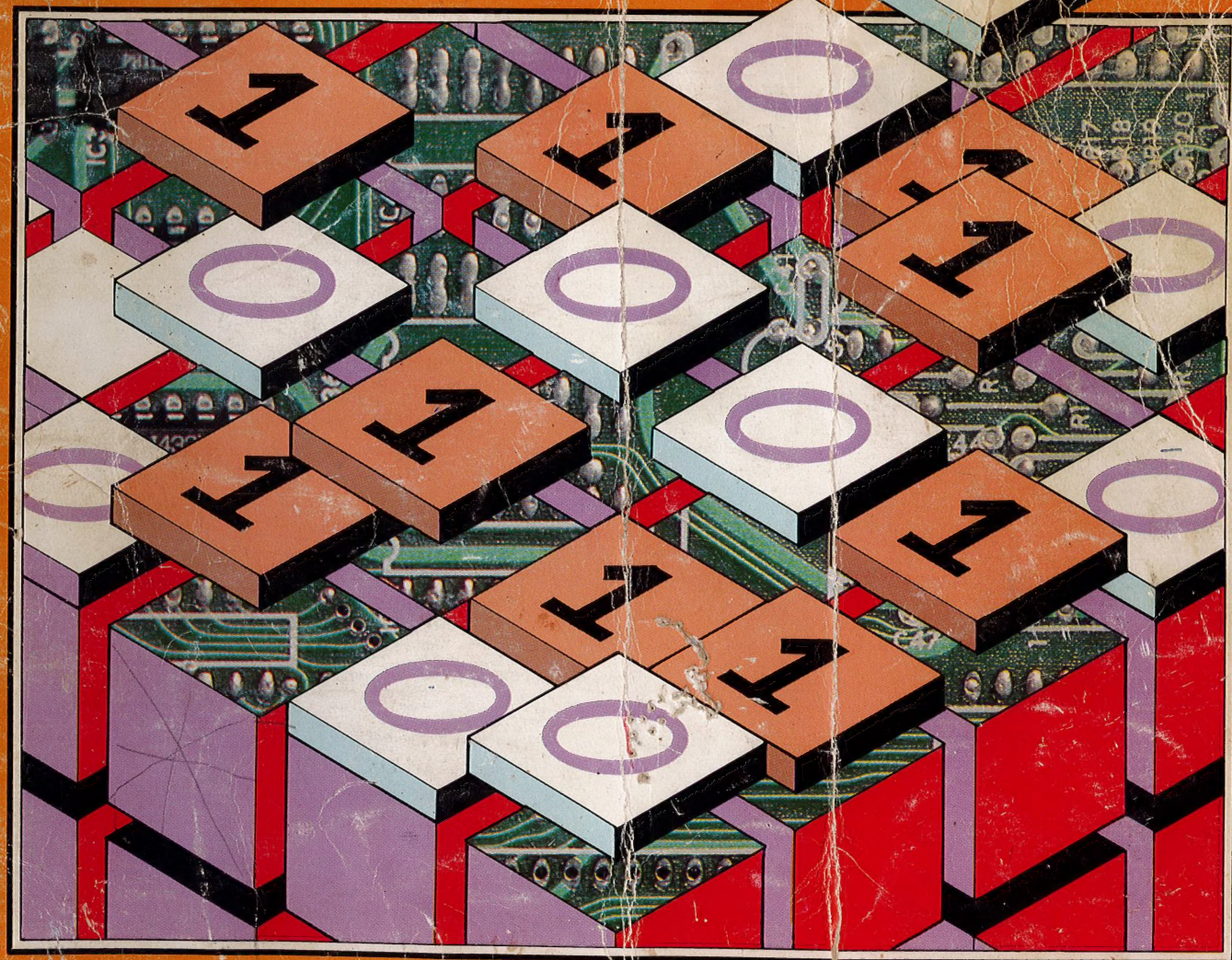
NZ \$11.50



## ASSEMBLY LANGUAGE PROGRAMMING

by Lewis Badham

Life after BASIC



A Your Computer Publication

# Everything a Hackers micro should be



- Make speech
- Make music
- Communicate
- Experiment
- And enjoy
- with microbee



## ASSEMBLY LANGUAGE PROGRAMMING FOR THE MICROBEE

by Lewis Badham

FEDERAL PUBLISHING COMPANY PTY LTD SYDNEY 1984

18  
78

National Library of Australia card  
number and ISBN 0 86405 080 1

Microbee is a trademark of  
Applied Technology Pty Ltd, and is  
used in this book by permission of  
the trademark holder.  
Z80 is a trademark of Zilog Inc.

Copyright © P.R.L. Badham and  
Federal Publishing Company, 1984.

All rights reserved. No part of  
this work may be reproduced in  
any form or by any means without  
the written permission of the  
publisher.

Published by the Electronics  
Division of Federal Publishing  
Company Pty Ltd., 140 Joynton  
Avenue, Waterloo NSW 2017  
Australia.

Managing Editor: Jamieson Rowe,  
Publisher: Michael Hannan.  
Design: Ali White,  
Production: Vernon, Rivers  
& Associates Pty Ltd.  
Distributed nationally by Gordon  
and Gotch Limited.

<b>CHAPTER 1</b>	•	<b>BASIC CONCEPTS</b> <i>What is Machine Language. Number systems. Binary numbers. Hexadecimal numbers.</i>
<b>CHAPTER 2</b>	•	<b>ESSENTIAL ELEMENTS OF A COMPUTER</b> <i>The Central Processing Unit. Random Access Memory. Input and Output devices.</i>
<b>CHAPTER 3</b>	•	<b>ASSEMBLY LANGUAGE INSTRUCTIONS</b> <i>The Z80 instruction set.</i>
<b>CHAPTER 4</b>	•	<b>WRITING AND RUNNING PROGRAMS</b> <i>Your first program. Using the Assembler. Using the Monitor. Saving and loading Source code and Object code.</i>
<b>CHAPTER 5</b>	•	<b>ASSEMBLER INSTRUCTIONS</b> <i>Labels. Pseudo-Ops.</i>
<b>CHAPTER 6</b>	•	<b>PROGRAMMING TECHNIQUES</b> <i>Messages. Arithmetic. Number conversions. Tables. Sorting. Inputting and displaying strings and numbers.</i>
<b>CHAPTER 7</b>	•	<b>GRAPHICS AND SOUND</b> <i>The Programmable Character Generator. LORES graphics. HIRES graphics. Moving graphics. Sound effects.</i>
<b>CHAPTER 8</b>	•	<b>DRIVING PRINTERS</b> <i>Output port control. Output to printers. ASCII screen dump. Graphics screen dump.</i>
<b>CHAPTER 9</b>	•	<b>MORE ARITHMETIC</b> <i>Negative numbers. Multiple precision. Decimal numbers.</i>
<b>APPENDICES</b>		
<b>Appendix 1</b>	•	<i>Decimal-Hex-Binary Conversion Table.</i>
<b>Appendix 2</b>	•	<i>ASCII Code.</i>
<b>Appendix 3</b>	•	<i>Z80 Instruction Set.</i>
<b>Appendix 4</b>	•	<i>Useful ROM Routines.</i>
<b>Appendix 5</b>	•	<i>PCG Character Address.</i>
<b>Appendix 6</b>	•	<i>Screen Location Addresses.</i>
<b>Appendix 7</b>	•	<i>Two's Complement Algebra</i>

# FOREWORD

Assembly Language Programming is a particularly relevant supporting text for the microbee. Unlike many other computers on the market, a very large number of the system and applications software for the microbee were developed using assembly language programming. This is a direct result of the 'hobby' background of the machine and has some very interesting implications.

In the USA a lot of software for personal microcomputers was developed using memory hungry compilers and other software tools. The result is that a lot of the US software requires a substantial amount of memory to run and, generally, the program execution time can be longer. Programs written in assembly language produce very compact object code and usually run very fast. The difference in the approaches may be seen in the version of Wordstar recently announced in ROM. This software has been extensively re-written to pack up the code but the resulting ROM space required is 40K. Wordbee, on the other hand, performs about 90% of the same functions and it fits into an 8K ROM with room to spare!

One real advantage to assembly language programmers is that they become very familiar with the actual operation of the microcomputer itself. This has produced a new breed of hardware/software literate programmers who as a consequence are quite fluent with the applications of the microprocessor as a tool. The disadvantage of this approach is that software is not easily ported to other processors and errors produced are usually bugs that 'crash' the programs outright. 'Debugging' assembled code is generally fast and a very satisfying experience. Programs developed under higher level languages such as compilers usually can be easily transported to other processors and also generate a 'softer' type of bug that may not be too serious but can often be very hard to find.

Assembly language programming has a strong traditional basis in the history of microprocessing and this book is highly recommended for those who wish to master this powerful and effective programming technique. I would like to express my personal thanks to Lewis Badham for the foresight to write such a book and bring to all one of the traditions of the not-so-ancient microprocessor era of days gone by.

Owen Hill,  
Managing Director.

Applied Technology.  
December 1984.

# INTRODUCTION

**T**his book is intended for owners of Microbee computers who have become reasonably competent BASIC programmers, and now wish to explore the mysteries of Assembly Language. It is assumed that you already have an Editor-Assembler fitted to your machine so that you may 'learn by doing'.

Although many articles and books have been written on Z80 assembly and machine language programming, the author believes that none adequately deals with the initial steps, so that they leave the prospective programmer confused and disheartened. This book endeavours to guide the uninitiated step by step through the early stages of assembly language programming.

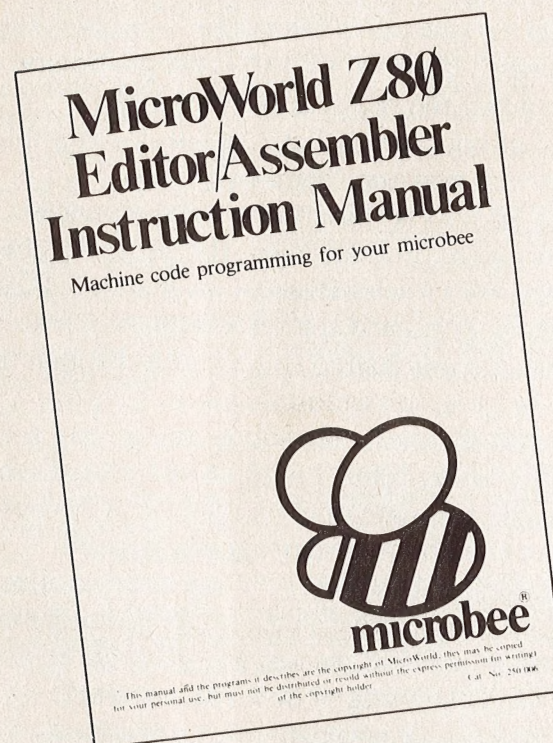
Although written specifically for the Microbee owner, it is also to a large extent applicable to any home computer which uses the Z80 CPU. In spite of starting at square one, by the end of the book you should be able to write mathematical programs, games programs with moving graphics, (Asteroids, Frogger, explosions, etc.), sound effects, and be able to drive a printer; all running many times faster than the equivalent BASIC programs. In addition, you will have a much better understanding of how the computer works, and will then be in a position to read more advanced texts on programming and operation.

The author acknowledges with gratitude the assistance of David Badham in supplying much of the material of Chapter 7, and that of Garry Epps for advice on some of the matters covered in Chapter 9.

Lewis Badham

# MicroWorld Z80 Editor/Assembler

A must for all microbee  
assembly language programmers



Available as a ROM Pack for  
16 and 32K microbees and also  
on disk for 64 and 128K microbees

Complete with manual all  
for the low cost of only **\$29<sup>50</sup>**

Manual only **\$14<sup>95</sup>**

When ordering please specify either:

- A. For 8,000 series 2 x 2732 ROMS
- B. For 9,000 series and above 1 x 2764 ROM
- C. 3.5" Diskette
- D. 5.25" Diskette

**APPLIED TECHNOLOGY RETAIL PTY LTD**

To order direct, phone (02)487-2711

## BASIC CONCEPTS

*Machine language, and the associated numbering  
systems and arithmetic — binary and hexadecimal —  
explained.*

When you type a BASIC program into your computer, the instructions you type are stored in the computer more or less in the same form as you type them. For example, if you type `PRINT 3 + 5`, the computer stores 131 3 + 5 (131 is the number code for `PRINT` in the Microbee). Now when you type `RUN`, a part of the computer called the BASIC INTERPRETER looks at the instruction and converts it into a series of *machine language instructions* which it then sends to the Central Processing Unit, or CPU.

The CPU is the heart of your computer, and in the Microbee it is a Z80. The Z80 CPU can only understand Z80 machine language instructions and does not understand BASIC instructions, or instructions for any other type of CPU. That is why it needs the BASIC INTERPRETER to interpret the BASIC instructions for it. When it receives the machine language instructions from the BASIC INTERPRETER it does what it is told, and in the example above, 8 appears on your monitor screen.

However, because the BASIC INTERPRETER has to convert each instruction into machine language, BASIC programs run comparatively slowly. What we are going to do is learn how to write programs directly in machine language so that the CPU can do what we want it to do without having to use the BASIC INTERPRETER. The result of this is that the program will run much faster, and incidentally take up less memory.

Strictly speaking, we're not really going to learn how to write programs in machine language itself, because this 'language' is actually just a string of numbers — and binary numbers at that. We'll explain about this shortly. Because true machine language is very messy, no-one bothers to use it anymore. Instead, 'machine language' programming is now done using *assembly language*, which is basically a symbolic version of the same thing. Assembly language is much easier to work with, because it uses easily-remembered little words called *mnemonics* to stand for the 'real' machine-language numbers. Then, when you've written the program using these mnemonics, the computer itself is used to translate them into the real machine language, using a program called an *assembler*.

You will find, as you progress through the book, that the instructions that the CPU can understand, which make up the machine language, are mostly very simple. For example, 'take this number and put it there', or 'add this number to that', or 'jump to another part of the program'. Because they are simple, to carry out the BASIC instruction above may take four, five or even more machine language instructions. This may seem tedious as you write your first programs, but as you gain experience you will start to think in terms of these simple instructions and you will find it quite easy.

### Number Systems

Before we go any further, it is necessary to understand how numbers are represented in the computer. Then we can study the CPU in more detail, and

how it interacts with the Random Access Memory, or RAM.

The system we commonly use to represent numbers is based on the number ten and is therefore called the decimal system (DEC is Latin for ten). To write numbers based on this system the Arabs invented ten symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. To count using these symbols we start at 0, and as we count we use the symbols in order until we come to 9. Having run out of symbols, we indicate the next number by 10, meaning one lot of ten plus no 'units'. Additional units are placed in the right hand position (11, 12, etc) until we run out of symbols again (19). We then change the symbol in the second position from 1 to 2, meaning 2 lots of ten, or 20.

Consequently, digits in the right hand position represent units, up to nine. Digits in the next position to the left represent 'tens', and digits in the next position to the left represent 'tens of tens', which we call hundreds, and so on. Thus, the number 1234 means

$$\begin{array}{r} \text{plus } 4 \\ \text{plus } 3 \times (10) \\ \text{plus } 2 \times (10 \times 10) \\ \text{plus } 1 \times (10 \times 10 \times 10) \end{array}$$

Of course, you are all very familiar with this, and are probably wondering why I am labouring the point. Well, computers don't work very well with decimal numbers, so all numbers in your computer are represented by a system based on the number two, called binary numbers or numbers in base two (BI is Latin for two).

### Binary Numbers

The binary system of numbers, as you can guess, uses only two symbols: 0 and 1. To count in this system we start, as in the decimal system, at 0, and the first thing we count is represented by 1, as in the decimal system. However, we have now run out of symbols, so we indicate the next number (which is two in the decimal system) by 10. This means one lot of two plus no units. The next number will be 11, meaning one lot of two plus one unit. The next number will be 100, meaning one lot of 'twos of twos', no 'twos', and no units. Consequently, as an example, the binary number 1011 means

$$\begin{array}{r} \text{plus } 1 \\ \text{plus } 1 \times (2) \\ \text{plus } 0 \times (2 \times 2) \\ \text{plus } 1 \times (2 \times 2 \times 2) \end{array}$$

which in the decimal system is

$$\begin{array}{r} 1 \\ + 2 \\ + 0 \\ + 8 \\ \hline 11 \end{array}$$

Putting this in a slightly different way, the value of the right hand position of a binary number is 1, of the next position is 2, the next is  $2^2$ , the next  $2^3$  and so on, just as the values of equivalent positions of a decimal number are 1, 10,  $10^2$ ,  $10^3$  and so on. Thus, in decimal, the digits in each position represent the following values:

$10^7$	$10^6$	$10^5$	$10^4$	$10^3$	$10^2$	10	1

In binary they represent:

(128)	(64)	(32)	(16)	(8)	(4)	(2)	
$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	2	1

Now have a look at Appendix 1 which is a table of decimal numbers up to 255 with their binary equivalents (ignore for a moment the column headed Hex). Note that all the binary numbers are shown with eight positions, even though the numbers up to 127 do not use all eight positions. *In these numbers the vacant positions are filled with 0s.* For instance 22 is 10110 in binary, but is represented as 00010110.

In fact this is the way all numbers are represented in your computer, that is, as binary numbers with eight positions. The digits in each position are either 1 or 0, and are called 'bits' (short for 'binary digits'), so the numbers are referred to as 'eight-bit binary numbers'. Each such number is called a 'byte'.

From now on we will refer to the right hand position of these numbers as bit 0, progressing leftwards to bit 7:

1	0	0	1	1	0	0	1
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

We can now say that for this particular number bit 0 is 1, bit 1 is 0, bit 2 is 0 etc. We can also say that bit 0 is 'on' or 'set' and bit 1 is off or 'reset'.

Note that the largest number that can be represented by an eight-bit binary number is 11111111, which in decimal is

$$\begin{array}{r} \text{plus } 1 \\ \text{plus } 1 \times (2^1) = 2 \\ \text{plus } 1 \times (2^2) = 4 \\ \text{plus } 1 \times (2^3) = 8 \\ \text{plus } 1 \times (2^4) = 16 \\ \text{plus } 1 \times (2^5) = 32 \\ \text{plus } 1 \times (2^6) = 64 \\ \text{plus } 1 \times (2^7) = 128 \\ \hline 255 \end{array}$$

(Actually it is easier to calculate  $2^8 - 1$  because the next number is 100000000 which is  $2^8$ ).

### Binary Arithmetic

If you convert the rules of addition, subtraction, multiplication and division which you use for decimal arithmetic into binary, the rules will work just the same.

For instance, to add two binary numbers, say

$$\begin{array}{r} 10010111 \\ + 01011010 \\ \hline 11110001 \end{array}$$

first add bits 0 together:

$$1 + 0 = 1$$

then:

bits 1    1 + 1 = 0 carry 1  
 bits 2    1 + 0 + carry = 0 carry 1  
 bits 3    0 + 1 + carry = 0 carry 1  
 bits 4    1 + 1 + carry = 1 carry 1  
 bits 5    0 + 0 + carry = 1  
 bits 6    0 + 1 = 1  
 bits 7    1 + 0 = 1

Just to convince yourself that the system works, look at Appendix 1 and you will find that you have done this sum:

$$\begin{array}{r} 151 \\ + 90 \\ \hline 241 \end{array}$$

The rules of multiplication also apply, but at this stage all we will say is that to multiply by ten in the decimal system you move the number one place to the left (by adding a 0 to the right hand side). In binary if we do this we *multiply by two*. For example:

$$\begin{array}{r} 01001011 \\ \times 2 \\ \hline 10010110 \end{array}$$

Check this in the table in Appendix 1 and you will find:

$$\begin{array}{r} 75 \\ \times 2 \\ \hline 150 \end{array}$$

### Hexadecimal Numbers

One other number system which we must understand is one that is based on the number sixteen, called hexadecimal (HEX is Greek for six). In this system, as you may have guessed, we need sixteen symbols, and we use the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. Here is a list of the first sixteen numbers in hexadecimal, and their equivalent binary and decimal values:

Hex	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

As before, when we run out of symbols the first time through, we indicate the next number by 10, meaning one lot of sixteen plus no units. By comparison with the decimal and binary systems, the symbols in each position of a four-hexadecimal-digit number represent the following values:

4096	256		
(16 <sup>3</sup> )	(16 <sup>2</sup> )	16	1

Don't forget that hexadecimal numbers can use the digits 0 to 9 and the letters A to F. So the hex number 3B5D means:

$$\begin{array}{r} D \\ \text{plus } 5 \times (16^1) \\ \text{plus } B \times (16^2) \\ \text{plus } 3 \times (16^3) \end{array} \quad \text{i.e.,} \quad \begin{array}{r} 13 \\ \text{plus } 5 \times (16^1) \\ \text{plus } 11 \times (16^2) \\ \text{plus } 3 \times (16^3) \end{array}$$

The rules of arithmetic apply, as with the decimal and binary systems. For instance,

$$\begin{array}{r} 3B5D \\ + 75C6 \\ \hline B123 \end{array}$$

$$\begin{array}{l} D + 6 = 3 \text{ carry } 1 \text{ (count E F 0 1 2 3)} \\ 5 + C + \text{carry} = 2 \text{ carry } 1 \\ B + 5 + \text{carry} = 1 \text{ carry } 1 \\ 3 + 7 + \text{carry} = B \end{array}$$

However, although it is sometimes useful to do this kind of simple arithmetic in hexadecimal, the importance of the hexadecimal system for computer users is its usefulness as a *shorthand for binary*. It becomes very tedious and time consuming to write eight-bit binary numbers every time we need to know exactly what the computer is doing. However, we can represent eight-bit binary numbers (bytes) by two symbol hexadecimal numbers which are much easier to remember. This is the only reason we use them.

Have another look at Appendix 1 which lists numbers from 0 to 255 in decimal, binary and hexadecimal form. Take any binary number and split it down the middle, so we have four bits on the left hand side and four bits on the right hand side (these four-bit parts are called 'nibbles'). Now inspect the hexadecimal equivalent. The left hand symbol is the hex equivalent of the left hand nibble, and the right hand symbol is the hex equivalent of the right hand nibble. For instance, decimal number 93 equals 01011101 in binary.

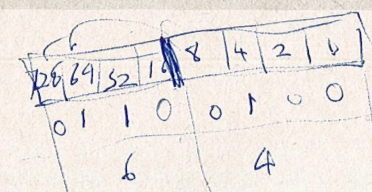
$$\begin{array}{r|l} 0101 & 1101 \\ \hline 5 & D \\ 0101 & \text{is } 5 \text{ hex} \\ 1101 & \text{is } D \text{ hex} \end{array}$$

So the equivalent of 01011101 is 5D. This works because each block of four bits represents a number from 0 to 15, which can also be represented by a single hex symbol.

The important thing to remember is that the *computer does not use hexadecimal numbers*. Their importance lies in the fact that *we* use them as a shorthand for the computer's number system — binary. You must admit, it's a lot easier to write 5D than 01011101 — and also less likely to result in errors.

### 16 Bit Binary Numbers

Although your computer can only handle one eight-bit binary number at any one time, it is necessary also to understand sixteen-bit binary numbers. The reason for this will become apparent in the next chapter. The principles are exactly the



100 =

same, and the largest number which can be represented in sixteen bits is 1111111111111111. By comparison with the eight-bit example, this represents decimal  $2^{16}-1$ , which is 65535.

Hexadecimal shorthand can be used with sixteen bits just as with eight-bit numbers. Split the number into blocks of four bits, and assign the equivalent hex symbol to each block. Thus:

binary	0011	0101	1000	1100
=	3	5	8	C
	or 358C hex			

Sixteen-bit numbers are called two-byte numbers and are represented, as we have seen, by four hex symbols.

### Representation of Hex Numbers

To avoid confusion between decimal numbers and hexadecimal numbers, it is necessary that hex numbers are always followed by H. So the examples we have just used should be written

5DH  
and 358CH

It is most important when writing assembly or machine language programs not to forget the H when using hexadecimal numbers, otherwise your number will be regarded as decimal.

# ESSENTIAL ELEMENTS OF A COMPUTER

*The central processing unit, random access memory, and input and output devices.*

**Y**our computer consists essentially of a Central Processing Unit, or CPU (which we mentioned briefly at the beginning of Chapter 1), a Random Access Memory, or RAM, an input device (the keyboard) and an output device (generally the monitor). We will look at these basic elements in turn, and discover how they interact with each other.

### The CPU

The Central Processing Unit is the heart of any computer, controlling all activities within it. The activities are regulated by a master clock which works rather like the conductor of an orchestra, dictating the timing of individual occurrences. Not only does the conductor ensure that things happen in the right order, but also determines the speed at which they occur. (Chopin's Minute Waltz could take half a minute or two minutes, depending on how fast the conductor waves his baton). So it is with the CPU in your computer. The Microbee uses a Zilog Z80 CPU, and two different 'clock' speeds have been used, the earlier machines using a speed of 2 MHz (two million cycles per second) and the later ones 3.375 MHz. This means that two (or 3.375) million separate events occur every second!

### The Registers

The Z80 CPU contains a number of *eight-bit registers* named A, B, C, D, E, H, L (see Fig. 2-1) each being capable of containing one eight-bit binary number, or byte. You can think of each of these registers as a box with eight compartments, each compartment capable of containing a '1' or a '0'. They each can therefore contain numbers in binary form up to 255, as we learned in Chapter 1.

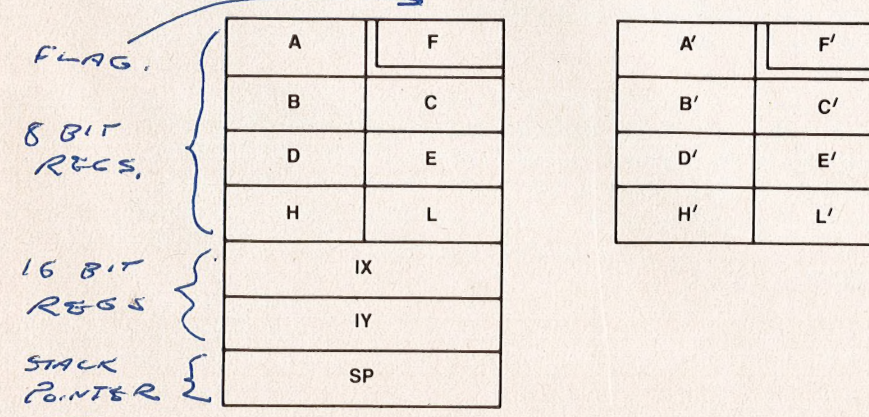
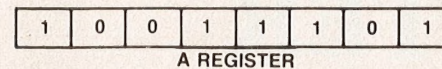


Fig. 2-1

To demonstrate how the registers are used, we will now introduce you to your first assembly language instruction,

LD A,157

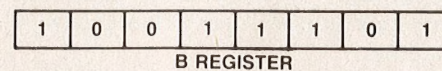
which means 'load the A register with the number 157' (in binary form, of course). This instruction will set the bits of the A register like this:



(check for yourself in Appendix 1). This number can also be transferred to another register with the instruction

LD B,A

which means 'load the B register with the number which is in the A register', so the B register will then also contain 157:



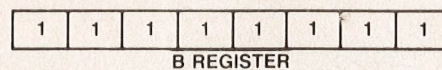
In the course of describing the operation of the CPU and the RAM we will introduce a number of other instructions. However a more detailed discussion of each instruction will be given in Chapter 3.

You will probably realise that for a lot of applications eight bits are simply not enough, because you will often want to use numbers greater than 255. For this reason some of the registers in the CPU can be used in *pairs*, namely B&C, D&E, H&L.

These register pairs can be regarded as 16-bit registers, and are therefore capable of containing numbers up to 65535, as we learned in Chapter 1. A typical instruction for register pairs is

LD BC,65535

in which case both the B register and the C register will have all bits set:



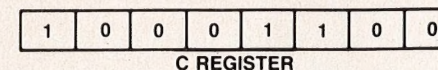
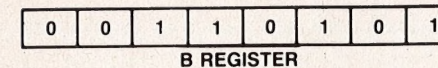
This instruction can also be written

LD BC,FFFFH

in which we have chosen to specify the number in hexadecimal form rather than decimal. As we pointed out in Chapter 1, we must place an H after the number to indicate this. As another example, using the hex number as discussed at the end of Chapter 1,

LD BC,358CH

will result in

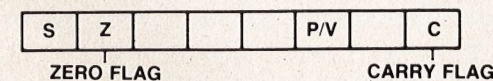


Note that the B register holds the 'high order byte', or bits 8 to 15 of the 16-bit binary number, while the C register holds the 'low order byte', or bits 0 to 7. Similarly, when D&E or H&L are used as pairs, registers D & H hold the high order bytes, while E & L hold the low-order bytes. The A register cannot be paired with any other register.

Before we leave the general purpose registers, if you refer back to Fig 2-1 you will see another set of registers designated A', B', C', D', E', H', L'. These are a complete set of *alternative* registers which may be exchanged for those currently in use. All or some may be exchanged, but you can only operate on one of the two alternatives (H or H' for example) at any one time. They are useful if you have numbers stored in all working registers and you need more 'storage space'.

### The Flag Register

The F register is a special purpose register and is called the Flag register. Its eight bits do not constitute a number, because the bits are used separately to indicate or 'flag' various *conditions*, or states-of-affairs inside the CPU. For example, the F register will indicate whether an arithmetic operation has resulted in a zero or negative answer. The Flag register looks like this:

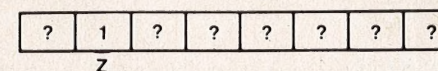


For the time being we will only concern ourselves with the Z, or Zero flag and the C, or Carry flag. The other flags will be discussed in Chapter 9 when we deal with negative numbers.

The Z flag is set (= 1) whenever an arithmetic or logical operation results in zero, and is reset (= 0) whenever such an operation does not result in zero. For instance, the two instructions

LD A,50  
SUB 50

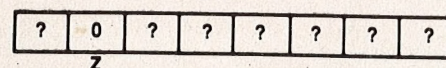
load 50 into the A register and then subtract 50 from the A register (the A register is understood in the SUB 50 instruction). The result, of course, is zero and the Z flag will be set *after the second instruction has been executed*. The Flag register will then look like this:



(the question marks mean that we don't know, or care at present, what the other bits are). On the other hand the instructions

LD A,50  
SUB 49

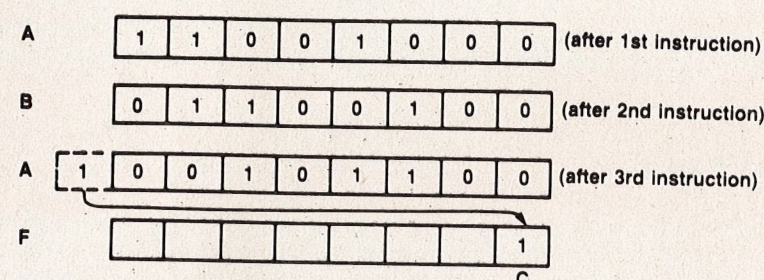
will reset the Z flag and the Flag register will look like this (after execution):



The C flag is set (= 1) whenever an arithmetic operation results in a register overload or carry-over. For instance,

```
LD    A,200
LD    B,100
ADD   A,B
```

will load 200 into the A register, load 100 into the B register, then add this number to the 200 already in the A register, like this

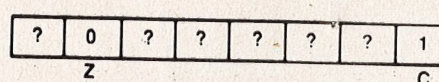


Because the result is greater than 255, when the addition is performed a Carry occurs from bit 7 of the A register. This Carry bit is placed in the C flag position of the Flag register, indicating that a register overload has occurred.

The C flag will also indicate when *subtraction of one number from another* has resulted in a negative answer. For instance, after the instructions

```
LD A,50
SUB 51
```

have been executed the Flag register will look like this:



Exactly why the C flag is set in these circumstances will be explained in Chapter 9.

The purpose of the flags is so that we can jump to a different part of the program depending on the result of some arithmetic operation, just like the BASIC

```
IF A > B THEN GOTO 250
```

For instance, the machine language instruction

```
JP Z,3A2CH
```

means 'jump to address 3A2C hex if the sum performed previously resulted in 0'. Likewise, the instruction

```
JP C,3A2CH
```

means 'jump to address 3A2C hex if the addition or subtraction performed previously resulted in a carry-over or a negative answer'.

If you are wondering what 'address 3A2C hex' means this will be explained shortly.

Before leaving the Flag register it must be emphasised that most, but not all, arithmetic instructions affect the flags, and some affect some flags and not others. Before relying on any instruction to condition or change a flag, consult Appendix 3 which lists the complete set of Z80 instructions including the flags affected by each instruction.

### Other Special Purpose Registers

The IX and IY registers of the CPU are 16-bit registers and can be used in the same way as register pairs to contain 16-bit or two-byte numbers. In addition, they have special properties which allow 'indexed addressing' which will be discussed later.

The Stack Pointer (SP) register is another special purpose 16-bit register. This register is used to control the operation of the Stack which will be discussed shortly.

### Memory

The second basic element of your computer, after the CPU, is the Memory. The memory is that part of the computer which contains (and remembers) what you have placed in it by keyboard, tape, disk etc. This information can be a program, data, or simply words, as when using the computer as a word processor.

Memory can be of two kinds, Random Access Memory or RAM, and Read Only Memory or ROM. In either case it consists of thousands of 'boxes' rather like the registers of the CPU, each capable of containing an eight-bit binary number, or byte.

The boxes, or memory locations, are numbered from 0 to 65535 and for each location this number is referred to as its *address*. In the case of RAM a one-byte number may be placed into, or read from, any memory location by specifying its address. The term Random Access refers to this fact, that any address may be randomly selected and not in any particular order. In the case of ROM, numbers cannot be placed into the memory locations, only read from them. The numbers have previously been placed in them in a permanent form.

If you now look at Fig 2-2 you will see a 'Memory Map' of the Microbee indicating the functions of the various parts of its memory. Notice that the addresses are listed in hex as well as decimal form. (In fact it is usually more convenient to use the hex form because the computer accesses memory in blocks of 1000 hex addresses).

The 'last' or highest address is, as we said before, 65535, or FFFF hex. This is because the addresses are two-byte numbers and, as we know from Chapter 1, the largest two-byte number is FFFF hex.

Each block of 1000 hex addresses contains  $1 \times 16^3$  or 4096 (decimal) addresses. This has become known loosely as 4000 or 4K.

Referring to Fig 2-2 again you will see that addresses 0 to 03FFF are listed as Scratch. This part of memory is used by the Microbee computer for various 'housekeeping' functions and is not available for use by the programmer.

Addresses from 0400H up are available to the machine language programmer, as we will discover in Chapter 4.

If you have a 16K Microbee, the last RAM address you can use will be 3FFFH and if you have a 32K machine you can use up to 7FFFH.

Addresses 8000H to BFFFH are occupied by the ROM of the BASIC Interpreter, and C000H to DFFFH by the EDASM ROM (if fitted). E000H to EFFFH are reserved for the Networking ROM, but may be used for other purposes.

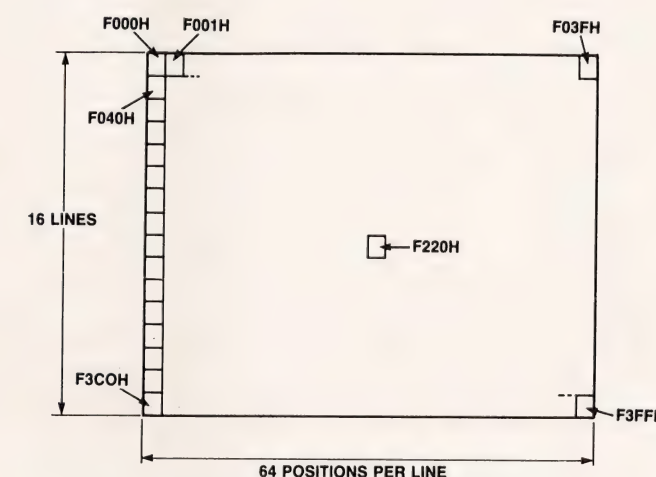
Addresses F000H to F3FFH are occupied by 'screen RAM' which is discussed below, and F800H to FFFFH are used by the Programmable Character Generator which will be discussed in Chapter 7.

**Fig. 2-2. Memory Map, 16K and 32K Microbee.**

ADDRESS		
HEX	DECIMAL	
0000 03FF	0	SCRATCH
0400 0900 1000	4096	START OBJECT CODE START BASIC PROGRAM START SOURCE CODE
2000 3000 3FFF	8192 12288 16383	RAM16K MICROBEE
4000 5000 6000 7000 7FFF	16384 20480 24576 28672 32767	EXTRA RAM 32K MICROBEE
8000 9000 A000 B000 BFFF	32768 36864 40960 45056 49151	BASIC INTERPRETER (ROM)
C000 D000 DFFF	49152 53248 57343	EDITOR ASSEMBLER (ROM)
E000 EFFF	57344 61439	NET (ROM)
F000 F3FF	61440 62463	SCREEN RAM
FFFF	65535	CHARACTER GENERATOR ROM & RAM

### Screen RAM

The 'screen RAM' addresses F000H to F3FFH (61440 — 62463 decimal) 'look like' any other addresses to the CPU, but they have a very special function. Each address represents a specific position on the video monitor screen. There are 16 horizontal lines of 64 positions each, making up a total of 1024 positions. See Fig 2-3. Address F000H represents the top left hand position, F001H the next position on the top line, and so on until F03FH which is the extreme right hand position of the top line. The next address, F040H is the left hand end position of the second line, and so on. The bottom right hand, or last position on the screen is F3FFH.



**Fig. 2-3. Microbee's Screen RAM**

### Using RAM

You cannot load numbers directly into RAM locations as you can into CPU registers; you must do it indirectly, by first loading the number into a CPU register and then loading the register contents into RAM:

```
LD A,72
LD (3000H),A
```

The first instruction, as you know, loads 72 into the A register. The second instruction means 'load RAM address 3000 hex with the number which is in the A register'. The brackets around the number mean 'RAM address with this number'.

After the above instructions have been executed, RAM address 3000 hex will look like this:

ADDRESS (HEX)	CONTENTS
2FFF	?
3000	0 1 0 0 1 0 0 0
3001	?

You can also do the same thing with register pairs;

```
LD BC,358CH
LD (3000H),BC
```

Hold it! We have just put a 16-bit number into register pair BC, and are now putting it into an eight-bit RAM address. How is this possible? Well, what happens is that the low-order byte goes into the specified address, 3000H, and the high-order byte goes into the next higher address, 3001H, like this:

ADDRESS (HEX)	CONTENTS
2 FFF	?
3000	1 0 0 0 1 1 0 0
3001	0 0 1 1 0 1 0 1
3002	?

or, in hex shorthand:

Address (hex,)	Contents
—	—
2FFF	??
3000	8C
3001	35
3002	??
—	—

You can also load a one-byte number into RAM like this:

```
LD HL,3000H
LD (HL),72
```

The first instruction *loads the number of the required address* into HL. Note that a register *pair* is necessary because addresses are two-byte numbers. The second instruction means 'load the RAM address identified by the number in HL with 72'. If you like you can think of HL as a *pointer* to address 3000H.

After these two instructions have been executed, RAM address 3000H will contain 72, which is the same result as above. This may seem a roundabout way of doing it, but it has many advantages which will become apparent later.

Yet another way to achieve the same result is

```
LD A,72
LD HL,3000H
LD (HL),A
```

where the last instruction means 'load the RAM address identified by the number in HL with the number in A'.

### The Stack

There is a special area set aside in RAM called the Stack. It is used for temporary storage of the contents of *register pairs* if, for instance, you want to use the register pair for some other purpose, and don't want to lose the number currently in it. A typical instruction to do this is:

```
PUSH HL
```

This instruction loads the contents of the HL register pair into the stack area of RAM exactly as in the examples above. The only difference is that you do not have to specify the address; the computer does it for you. To recover your stored two-byte number, you use the instruction

```
POP HL
```

This loads the stored number back into the HL register pair, where it originally came from. On the other hand, you can use

```
POP BC
```

in which case the number is loaded into the BC register pair. Thus the two instructions

```
PUSH HL
POP BC
```

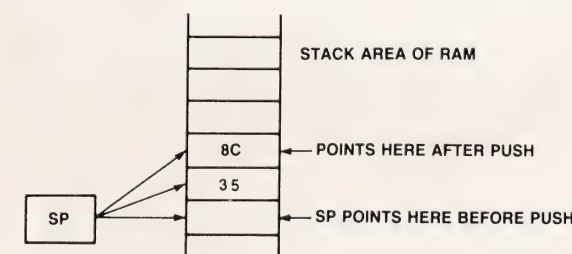
are a simple way of transferring the contents of the HL register pair to the BC register pair: there is no direct instruction to do this.

The operation of the stack is controlled by the Stack Pointer register (SP) of the CPU which we mentioned previously. The number in the SP register identifies the RAM addresses to be used.

To illustrate the operation of the stack, the instruction

```
PUSH HL
```

will place the byte in H in one address and the byte in L in the next *lowest* address. If HL contained 358CH the stack area of RAM would look like this after the push instruction had been executed:



The SP register is decremented (ie reduced by one) twice during a PUSH operation, and points to the last address used which then becomes the 'top of stack'.

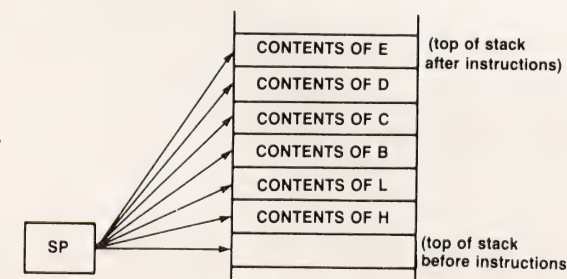
The instruction  
POP HL

takes the byte from the top of stack and places it into L, and the byte from the next *highest* address and places it into H. The SP register is incremented twice during a POP operation and points to the address above the top of stack after the operation.

If you keep PUSHing register pairs on to the stack you will automatically load progressively lower RAM addresses with the contents of the register pairs. For example:

```
PUSH HL
PUSH BC
PUSH DE
```

will result in



If we subsequently want to load the register pairs with their original numbers, we must POP them in reverse order like this:

```
POP DE
POP BC
POP HL
```

This method of operation is known as a LIFO system — 'Last In First Out'. If you like, you can think of it as a pile of books — you can only get to the last book you put on the pile.

### The Stack and Subroutines

The stack has another important use in relation to machine language subroutines. The equivalent of a BASIC 'GOSUB' statement in machine language is CALL. For example,

```
CALL 2DE3H
```

means jump to the subroutine which starts at address 2DE3 hex. The difference between this and a JP instruction is that the CPU has to remember at what part of the program the CALL instruction was encountered, so that at the end of the subroutine it can return (as in a BASIC GOSUB). The stack is used to store the address of the next instruction following the CALL instruction.

### Input Devices

The normal input device for your computer is the keyboard. The keyboard, as you know, has keys corresponding to letters of the alphabet, both capital and small (upper and lower case), numbers, and special characters such as +, minus, brackets, commas, full stops etc.

Because the computer can only understand numbers, each one of these keys has been allocated a number between 0 and 127 in accordance with the 'American Standard Code for Information Interchange', called ASCII for short. This code is reproduced in Appendix 2 which gives the code number in both decimal and hex form. The numbers from 32 to 126 represent printable characters (32 is SP, which will print a space!) while the others are mostly 'control' characters such as back space, tab, line feed, etc.

In addition to the keyboard the Microbee can also accept input from a cassette recorder, or through its parallel or serial ports, perhaps from a joystick controller or through the telephone lines via a MODEM.

### Output Devices

The normal output device is the screen of your monitor, or modified TV set. This, of course, enables you to see what you have typed, and also to see the results of running a program, whether it is a video game or a mathematical calculation. Output can also be sent to a cassette recorder for saving programs.

Another common output device is a printer, and on a Microbee a printer can be driven from either the parallel or serial port. The printer enables you to obtain a 'hard copy' of your program listings and runs. If you have a printer with graphics capability you can also obtain a picture of anything you have on the screen ('screen dump') and in Chapter 8 we will see how to do this with a machine language program.

As with input devices, all output to a monitor screen or printer must be in ASCII code.

# ASSEMBLY LANGUAGE INSTRUCTIONS

## The Z80 instruction set

The last chapter introduced a few Z80 instructions while describing the functions of the CPU and RAM. Now we shall look in detail at all of the most commonly used instructions, describing what they do and how to use them. They will be listed in *mnemonic* form (eg LD A,B) which makes it easy to remember what they do. As we noted before, this is 'Assembly Language', and the assembler in your Editor-Assembler (EDASM) package automatically converts this into true machine language. Just to confuse you, assembly language is also called *source code*, and machine language is also called *object code*. For example, the assembler will convert the instruction LD A,B into the number 78H, so 78H is the machine language or object code for that instruction:

Mnemonic	
OR Assembly Language	Machine Language
OR Source Code	OR Object Code
LD A,B	78H

From now on we will use the terms 'source code' and 'object code'. A complete list of all Z80 instructions showing both source code and object code is given in Appendix 3. You will note that the object code for many instructions, like the one above, consists of only one byte, while others are up to four bytes long.

In examining the more common instructions, it is convenient to separate them into types, as follows:

- Moving data
- Arithmetic
- Logic
- Bit manipulation
- Jumps and Calls
- Pushes and Pops
- Exchanges
- Input and Output

Within each type, typical instructions only will be discussed, and reference to Appendix 3 should then identify all other instructions of that type.

### Moving Data (Load) Instructions

The instructions in this group LOAD numbers into CPU registers or RAM addresses. The numbers are either actual numbers which you nominate, or are contained in other registers or RAM addresses.

It is important to understand that any number which is in the *destination* is

replaced by the new number, but the number in the *source* is not affected. For example,

```
LD  A,B
```

loads the A register with the number in the B register, *replacing what was in the A register but leaving the B register unaltered*. Similarly,

```
LD  (1234H), A
```

loads RAM address 1234H with the number in the A register, replacing what was in that address, but leaving the A register unaltered.

Putting this another way, after an LD instruction, both destination and source contain the same number.

Where the following instructions specify numbers, one-byte numbers will be indicated by n and two-byte numbers by nn. As explained before, these numbers can be written in decimal or hex form, but hex numbers must be followed by 'H'. Of course, where a number is loaded directly into a single register, this will have to be a one-byte number (max 255), and to a register pair (or IX, IY, SP), a two-byte number (max 65535).

Don't forget that *addresses* are two-byte numbers and must therefore be loaded into register pairs, but the *contents of an address* will be a one-byte number.

Where the following instructions contain a bracketed register or register pair, for example (HL), the description refers to 'address HL'. This means 'the RAM address which is identified by the number in HL'. With the IX and IY registers an offset may be specified which is added to the identified address. For example, if IX contains 2000H, (IX + 1) means address 2001H. However, note that (IX) is also valid, meaning address 2000H.

Here are the various kinds of Load instruction:

LD A,n	Load A register with number n
LD A,B	Load A register with number in B register
LD A,(nn)	Load A register with number in address nn
LD B,(HL)	Load B register with number in address HL
LD B,(IX + d)	Load B register with number in address IX + d
LD BC,nn	Load BC register pair with number nn
LD BC,(nn)	Load BC register pair with number in address nn
LD (nn),A	Load address nn with the number in A register
LD (HL),n	Load address HL with number n
LD (HL),A	Load address HL with number in A register
LD (nn),BC	Load address nn with number in BC register pair (C $\rightarrow$ nn, B $\rightarrow$ nn + 1)
LD (IX + d),n	Load address IX + d with number n
LD (IX + d),A	Load address IX + d with number in A register

There are two special load instructions which will load one block of consecutive RAM locations from another block of consecutive RAM locations. For instance, you can load the contents of addresses 3000H — 3040H into addresses 4000H — 4040H. You can do this from the bottom up (using LDIR) or from the top down (using LDDR). The respective start addresses must first be loaded into the HL and DE register pairs (HL contains the *source* address and DE the *destination* address) and the number of locations involved must be loaded into the BC register pair. Thus, for the example above:

```
LD  HL,3000H
LD  DE,4000H
LD  BC,41H
LDIR
```

Or we could have used the LDDR instruction by loading HL with 3040H and DE with 4040H. Here is a summary of the two:

LDIR — Load address DE with number in address HL. Increment HL and DE. Decrement BC and repeat until BC = 0

LDDR — As above except that HL and DE are decremented.

In most of the above typical instructions the registers shown may be replaced by other registers, and the register pairs may be replaced by other register pairs, or by IX, IY or SP. However, this is not always the case, so before you use any instruction, check Appendix 3 to make sure it is valid.

### Arithmetic Instructions

Arithmetic performed by the Z80 is limited to Add or Subtract instructions, mostly performed on the contents of the A register or HL register pair. They can either include the contents of the Carry flag or ignore it. Special add and subtract instructions are Increment (add one) and Decrement (subtract one). In all cases the register of RAM location affected by the instruction is left containing the result. For example, the instruction

```
ADD A,B
```

leaves A equal to A + B, and B unaffected. In other words, this instruction means 'add the contents of B to the contents of A, but leave B unaffected'.

Consult Appendix 3 to check if flags are affected for all instructions.

ADD A,n	Add to number in A register the number n
ADD A,B	Add to number in A register the number in B register
ADD A,(HL)	Add to number in A register the number in address HL
ADD HL,BC	Add to number in HL register pair the number in BC register pair
ADD A,(IX + d)	Add to number in A register the number in address IX + d

The above typical instructions are also available in an Add with Carry instruction, eg.

```
ADC A,n      Add to number in A register the number n plus
              contents of C flag
```

Most of the above typical forms are also available in a Subtract with Carry form, eg.

SBC A,B Subtract from number in A register the number in B register plus contents of C flag ie:  $A-(B+C)$

Subtract without Carry instructions are only available for the A register in the following typical forms:

SUB n Subtract from number in A register the number n  
 SUB B Subtract from number in A register the number in B register  
 SUB (HL) Subtract from number in A register the number in address HL  
 SUB (IX + d) Subtract from number in A register the number in address IX + d

INCrement and DECrement are special add and subtract instructions and are available for all registers and register pairs:

INC B Add one to number in B register  
 INC BC Add one to number in BC register pair  
 INC (HL) Add one to number in address HL

The above typical forms are also available in DECrement instructions, eg.  
 DEC B Subtract one from number in B register

Note that the INC and DEC instructions performed on one-byte numbers (eg INC B or INC (HL)) affect the Z flag *but not the C flag*. INC and DEC instructions on two-byte numbers (eg INC BC) *do not affect any flags*.

Another special subtract instruction is Compare. In this case the nominated number is subtracted from that in the A register, *but the A register is not affected*. The Z and C flags are set according to the result. Typical Compare instructions are:

CP n Compare number n with number in A register (A-n)  
 CP B Compare number in B register with number in A register (A-B)  
 CP (HL) Compare number in address HL with number in A register (A-(HL))

Most of the above instructions are also available with other registers and register pairs, but, as before, check in Appendix 3 to ensure that any particular instruction is valid.

Two Complement instructions are also available. A complement instruction changes any 0 to 1 and any 1 to 0.

CPL Complement A register  
 CCF Complement Carry Flag

### Logical Instructions

There is a group of instructions which performs logical AND, OR and XOR operations with the A register. Logical operations are explained below, and the relevant instructions described.

#### AND

A logical AND operation performed on two bits will result in a 1 only if *both* bits are 1, otherwise the result will be 0. The instruction

#### AND B

performs logical AND operations on corresponding bits of the A register and the B register, and places the result in the A register. For example, if the A and B registers contained

```

A      11110000
B      10101010
      11111111
      11111111
  
```

then AND B would produce A 10100000

#### OR

A logical OR operation performed on two bits will result in a 1 if *either or both* bits are 1, otherwise the result will be 0. As before, the instruction

#### OR B

performs logical OR operations on corresponding bits of the A and B registers and places the result in the A register. Using the same numbers as before,

```

A      11110000
B      10101010
then OR B would produce A 11111010
  
```

#### XOR

A logical XOR (Exclusive OR) operation performed on two bits will result in a 1 if *either* bit is 1, *but not both*, otherwise the result will be a 0. Again, the instruction

#### XOR B

performs logical XOR operations on corresponding bits of the A and B registers and places the result in the A register. Using the above numbers again,

```

A      11110000
B      10101010
then XOR B would produce A 01011010
  
```

The following general types of logical instructions are available:

AND n Perform AND operations between A register and the number n and place result in A register

AND B Perform AND operations between A register and B register and place the result in the A register

AND A Perform AND operations between A register and A register (!) and place result in A register. (This instruction can be used to set flags without changing the A register)

AND (HL) Perform AND operations between A register and number in address HL and place result in A register

AND (IX + d) Perform AND operations between A register and number in address IX + d and place result in A register

All logical operations are performed with the A register, and the result placed in the A register. Any single register can be used instead of B in the AND B type of instruction.

Similar instructions to the above are available for OR and XOR operations. As before, check Appendix 3 for the complete list.

Note that all the logical instructions affect the Z flag according to the result, *but always reset the C flag to 0*. For this reason the instructions AND A or OR A are frequently used to zero the C flag without affecting A (to prepare for a SBC instruction, for instance, if a subtract without Carry is required). Likewise, XOR A is frequently used simultaneously to zero A and the C flag, to 'wipe them clean' before an arithmetic operation.

### Bit Manipulation Instructions

So far we have encountered instructions which apply only to complete bytes. There are a number of instructions which enable us to manipulate the individual bits within bytes.

SET instructions will set (make equal to 1) a nominated bit in any of the single registers, or RAM addresses identified by HL, IX + d, or IY + d. Typical of this type are:

SET 4,B      Set bit 4 of the B register

SET 4,(HL)      Set bit 4 of number in address HL

SET 4,(IX + d)      Set bit 4 of number in address IX + d

RES instructions (RESet) are available in the same format as the SET instructions, and will RESet (make equal to 0) the nominated bit in the register or RAM address.

The SET and RES instructions are not available for the Flag register. However you can set the Carry flag with

SCF      Set Carry Flag (to 1)

and then reset it with CCF.

BIT instructions are available in the same format as SET and RES instructions, and will test the nominated bit without changing it, setting the Z flag if it is 0, and re-setting it if it is 1.

SHIFT and ROTATE instructions form a family which move all the bits in a register or RAM location one place to the left or right. The bit which is pushed out either goes into the Carry flag or is 'rotated' to enter the empty position. The operation of each instruction will be explained diagrammatically.

SLA D Shift Left Arithmetic D register



0 is placed in bit 0.

For example, if the D register contained D 00001111, the above instruction would produce

C flag = 0      D 00011110

You may remember from Chapter 1 that this will multiply the number in the D register by 2.

Other typical SLA instructions are

SLA (HL)      Shift Left Arithmetic number in address HL

SLA (IX + d)      Shift Left Arithmetic number in address IX + d

Other Shift and Rotate instructions are shown in Fig 3-1. They are available in the same formats as the SLA instruction. The diagrams have been simplified, but remember that the arrow means that each bit is moved in the specified direction one position.

The SLA instruction is included again for completeness.

### Jumps and Calls

A JP instruction will cause the CPU to Jump to the RAM address specified, that is, take its next instruction from that address. For example,

JP 12ABH

will produce a jump to address 12AB hex. This address will contain the object code for some instructions which will consequently be executed next. This is similar to a BASIC GOTO instruction, except that a RAM address is used to locate the next instruction, rather than a line number.

Jumps may also be made to RAM addresses identified by HL,

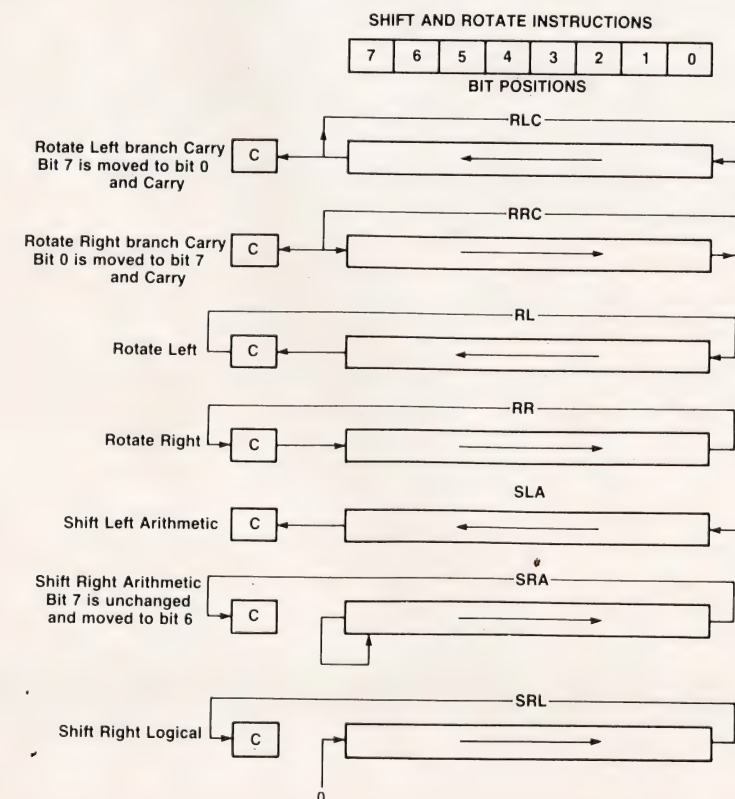


Fig. 3-1

IX, or IY. So the two typical forms of jump instructions are:

JP nn Jump to address nn  
JP (HL) Jump to address HL

The above instructions are *unconditional* jumps which occur whatever the condition of the flags. *Conditional* jumps are also available, as follows:

JP NZ,nn Jump to address nn if Z flag not set  
JP Z,nn Jump to address nn if Z flag set  
JP NC,nn Jump to address nn if C flag not set  
JP C,nn Jump to address nn if C flag set  
JP M,nn Jump to address nn if M flag set  
JP P,nn Jump to address nn if P flag set

JR instructions (Jump relative) differ somewhat from JP instructions in that they cause the Z80 to jump *forwards* or *backwards* by a specified number of addresses, rather than to a specific location. The end result is the same, but the object code is only two bytes instead of three. The jump is limited to +127 or -128 addresses, counting from the end of the instruction itself.

Typical instructions are:

JR n Jump n addresses (n can be + or -)  
JR NZ,n Jump n addresses if Z flag not set

There is a special JR instruction which will jump repeatedly for a specified number of times. The required number of times first must be loaded into the B register. The DJNZ n instruction counts the number of times jumped by decrementing the B register and testing for zero each time.

DJNZ n Decrement B register. If this does not set Z flag then jump n addresses.

The CALL instruction is similar to a BASIC GOSUB. It is used for CALLing a subroutine stored in some other memory area. At the end of the subroutine a RET instruction will RETURN the Z80 to the instruction in the main program immediately following the CALL (like the BASIC RETURN).

Both unconditional and conditional CALLs are available:

CALL nn Go to subroutine at address nn  
CALL NZ,nn Go to subroutine at address nn if Z flag not set

Unconditional and Conditional RETURNS are also available:

RET Return from subroutine  
RET NZ Return from subroutine if Z flag not set

### Pushes and Pops

PUSH and POP instructions were discussed in Chapter 2 when we were examining the operation of the stack. Remember that a PUSH instruction places the two-byte number which is in the specified register pair on to the top of the stack. A POP instruction takes the two-byte number at the top of the stack and places it into the specified register pair. You can also PUSH and POP AF (the A register and Flag register), although strictly not a register pair, and the two-byte index registers IX and IY.

The typical instructions are:

PUSH HL Place number in HL register pair on top of stack.  
POP DE Place number at top of stack into DE register pair.

Note that, as with LD instructions, the PUSH instruction leaves the nominated register pair unaltered.

### Exchange Instructions

When we were discussing the CPU in Chapter 2 we mentioned that an alternative set of registers, A', F', B', C', D', E', H', L' was available. The following instructions EXchange the currently used registers with the alternative registers. The alternatives then become the current registers and all instructions will then apply to these. On another EXchange, the registers are again swapped, and will have retained the numbers originally contained in them.

EX AF,AF' Exchange A and F with alternative A and F.

EXX Exchange BC, DE, HL with alternatives.

An exchange instruction is also available to swap the contents of the DE and HL register pairs.

EX DE,HL Load DE register pair with number in HL, and vice versa.

The top of the stack may also be exchanged with some registers.

EX (SP), IX Load addresses SP and SP + 1 with number in IX register and vice versa.

### Input and Output Instructions

These instructions enable you to send or receive information (bytes) to or from a specified PORT. The ports which you can use are listed in the Microbee Users Manual. Applications will be discussed later.

IN A,(n) Load A register with number in port n  
OUT (n),A Load port n with number in A register  
IN A,(C) Load A register with number in port identified by C register  
OUT (C),A Load port identified by C register with number in A register

The last two instructions are available with registers other than the A register.

### Conclusion

The preceding description of typical Z80 instructions covers most of those available, and all that you will need within the scope of this book. Appendix 3 is a complete list of all the Z80 instructions with the object code and flag effects of each.

## WILDCARDS

Volume One

A potpourri of application notes and tips  
for the  
MicroBee Personal Computer

The WILDCARDS series  
take over where the  
user's manual ends.  
They enable the user to  
get more out of the  
MicroBee.  
They contain a wealth of  
information for  
beginners or experts.

## WILDCARDS

Volume Two

A potpourri of application notes and tips  
for the  
MICROBEE™  
personal computer

## WILDCARDS

Volume Three

A collection of programs, tips and techniques  
for all models of the  
MICROBEE™  
personal computer

Available from MicroBee Technology Centres & Dealers.

# WRITING AND RUNNING PROGRAMS

*How to write your first program, use the assembler and monitor, and save and load source and object code.*

This chapter shows how to write and run a simple machine language program so that you can see how the instructions in the previous chapter are used, and familiarise yourself with the operation of the Editor-Assembler.

The first question, of course, is what is an Editor-Assembler. The EDASM package in your MicroBee is actually three packages in one, an Editor, an Assembler and a Monitor. What does each do?

When you have written your program it will be in source code (mnemonic form, or assembly language) as we mentioned at the beginning of the previous chapter.

The Editor simply controls the operation of typing in the source code, allowing you to make corrections, insert extra lines, list the program, save to cassette, and so on, much the same as the equivalent operations for a BASIC program. However it uses a different set of instructions which are listed in your Editor-Assembler manual. Although the list is extensive you can actually manage with a very few simple instructions which we will introduce in this chapter. As you become more familiar with writing and running programs you will be able to understand how to use the other instructions to save time and perform special tasks.

After you have typed in your program you can use the Assembler automatically to convert the source code instructions to object code bytes, which it loads into a part of memory which you specify. After that you then execute (run) your program.

The Monitor enables you to inspect any RAM address to see what it contains and, if you wish, to change it. This is a very useful, even essential, capability for understanding and de-bugging programs.

Unfortunately the word monitor is also used to refer to the Visual Display Unit (VDU), or screen. When the screen is intended in this book the word will be printed with a small m, and when the Monitor package in the Editor-Assembler is intended it will be printed with a capital M.

### Your First Program

Now you are ready to write and run your first program. It will be a very simple one, but will introduce some basic principles.

Let us write a program to display the number 7 in the bottom right hand corner of the screen. The equivalent BASIC program to do this would be:

```
100 CLS
110 CURS 64,16
120 PRINT "7"
130 END
```

NOTE: For our machine language version, we will first find how to place the 7 in the required position on the screen, and we will look at how to clear the screen later.

You may remember from Chapter 2 that screen positions are allocated RAM addresses F000H to F3FFH, the latter being the bottom right hand position, so all we have to do is place a 7 in this address. We also know from Chapter 2 that we cannot load a number directly into a RAM address, but must first load it into a register. So let us write

```
LD A,7
LD (F3FFH),A
```

There is one thing wrong with this. When we were discussing input and output devices in Chapter 2 we noted that all information has to be transferred to and from these devices in ASCII code. The monitor screen is an output device, so the information placed in screen RAM must be in ASCII form. Consequently, to place a '7' on the screen we must use the ASCII code for 7, which is 55 (see Appendix 2). So, let us now rewrite our program like this:

```
LD A,55
LD (F3FFH),A
```

That should work, so we will now type it, assemble it and execute it.

### Using The Editor

We must first get 'into' the Editor-Assembler program in our Microbee, as follows.

If you have a 16K Microbee, type:

**LOCK** The Editor-Assembler will only accept capital letters

EDASM **RETURN** The screen will display 'Memory Size?'

4000 **RETURN**

Your 16K Microbee has 4000 hex RAM locations. (See Chapter 2).  
The EDASM assumes your answer is in hex so an H is not necessary.

Editor-Assembler

\*

will now appear on a blank screen with a flashing cursor, indicating that it is ready to accept your program.

If you have a 32K Microbee, replace your response to 'Memory Size?' with

8000

Now type: (I'll answer your questions in a minute)

```
I RETURN
TAB ORG TAB 400H RETURN
TAB LD TAB A,55 RETURN
TAB LD TAB (OF3FFH),A RETURN
```

**TAB** **RET** **RETURN**

**TAB** **END** **RETURN**

**BREAK**

The screen should now display:

Editor-Assembler

```
*I
00100      ORG      400H
00110      LD       A,55
00120      LD       (OF3FFH),A
00130      RET
00140      END
00150
*
```

Now to answer your questions.

I stands for Insert. Line numbers will automatically start at 100 and step by 10. This works exactly the same way as the BASIC AUTO command except that the I command *must* be used to insert lines in the EDASM, whereas the BASIC AUTO command is optional.

If you make a mistake in typing, say in line 120, type

R120 **RETURN** (R stands for Replace)

Re-type the line, including the RETURN at the end. Ignore the message 'no room between lines'.

What does ORG 400H mean? ORG stands for ORiGin, and is an instruction to the Assembler to place the object code version of the program into RAM starting at address 400H.

The TAB key spaces the program neatly in columns to make it more readable. Spaces are also a requirement of the Assembler.

Note that we have left a TAB space between the line number and the instructions. Later we will use the space to insert names for some of the lines. These names are called LABELS and are used by the Assembler for JUMP instructions in the same way as line numbers are used in BASIC for GOTO instructions.

At line 120 we typed a 0 in front of F3FFH. Why? Because the Assembler would think that the F was the first letter of a LABEL and become confused. Consequently, where an address in hexadecimal starts with a letter (A, B, C, D, E, or F), you *must* put a 0 in front of it.

The RET instruction tells the CPU (when it ultimately executes the program) that it has finished executing the program, and should return to the Editor-Assembler.

The END statement, like the ORG statement, is an instruction to the Assembler (not the CPU) and simply tells it to stop the assembly process because it has come to the end of the program.

Note that the RETURN key is pressed after the END statement, to incorporate it into the program. BREAK then exits from the Insert mode, and the asterisk tells you the Editor is ready for your next command.

Before we do anything else, let us review what we have done. We have typed a list of instructions in mnemonic or source code form, which are now stored in memory. The area in RAM used for this purpose in the Microbee is from 1000H.

To understand what this means, if you looked at this part of memory you would find the contents of each address as shown in Figure 4-1.

ADDRESS (HEX)	CONTENTS		
	HEX	DEC	ASCII
1000	64	) 100	
1001	00	)	
1002	09	9	
1003	09		HT
1004	4F		O
1005	52		R
1006	47		G
1007	09		HT
1008	34		4
1009	30		0
100A	30		0
100B	48		H
100C	6E	) 110	
100D	00	)	
100E	08	8	
100F	09		HT
1010	4C		L
1011	44		D
1012	09		HT
1013	41		A
1014	2C		,
1015	35		5
1016	35		5
1017	78	) 120	
1018	00	)	
1019	0E	14	
101A	09		HT
101B	4C		L
101C	44		D
101D	09		HT
101E	28		(
101F	30		0
1020	46		F
1021	33		3
1022	46		F
1023	46		F
1024	48		H
1025	29		)
1026	2C		,
1027	41		A
1028	82	) 130	
1029	00	)	
102A	04	4	
102B	09		HT
102C	52		R
102D	45		E
102E	54		T
102F	8C	) 140	
1030	00	)	
1031	04	4	
1032	09		HT
1033	45		E
1034	4E		N
1035	44		D
1036	FF		
1037	FF		

Addresses 1000H and 1001H contain the line number, in *hex* form, except that the two bytes have been transposed. That is, the low byte has been stored first (in 1000H) and the high byte next (in 1001H). So the line number is 0064H, which is 100 decimal. The next byte, 09 in 1002H is the number of bytes to the end of that line. The following bytes are the ASCII codes for the source code. Read down the right hand column and you will find exactly what you typed (HT stands for Horizontal Tab). The two FF's at the end of the program serve to indicate to the Assembler just that - the end of the program.

So the numbers, letters, spaces, commas etc, that you have typed are actually stored in memory using the ASCII code, starting at address 1000H.

You are probably getting impatient as, after all, we still have not run this very simple program. However, we have one more job to do.

### Using The Assembler

As explained previously, the Assembler automatically converts our source code into object code and loads it into RAM starting at the selected ORiGin. In our example the first instruction (LD A,55) will be converted to 3E37 (hex is understood). Check this for yourself in Appendix 3. The 3E represents LD A, and the 37 is the hex equivalent of 55 decimal. This is a two-byte instruction, so the first byte, 3E, will be placed in address 400H and the second, 37, in 401H. Each instruction in the program will be converted in turn as above and the resultant object code bytes placed into succeeding addresses.

To initiate the assembly process, type

A/WE RETURN

This stands for Assemble and Wait on Error. If, during the assembly process the Assembler comes across an instruction it does not understand, it will wait at the error and tell you about it! If this happens, simply re-type the offending line using the R instruction, and then try another assembly.

If all is well your screen will look like this:

```
*A/WE
0400          00100      ORG      400H
0400 3E37      00110      LD       A,55
0402 32FFF3     00120      LD       (0F3FFH),A
0405 C9         00130      RET
0000          00140      END
00000 Total errors
```

\*

The first column of the assembled program (ie after \*A/WE) lists the address of the *first* byte of each instruction. As we expected, address 400 contains 3E. The next byte, 37, is in 401, 402 contains the first byte of the next instruction, 32. The next two bytes are in 403 and 404, and they form the address part of the LD (0F3FFH),A instruction. Note that the leading 0 has been dropped and that the *two bytes of the address have been transposed*. 405 contains C9, the object code for the RET instruction. Satisfy yourself that all the instructions have been correctly converted from source code to object code, and note that all addresses and object code are in hex, but the H has been dropped.

Now the big moment has arrived. You have a program in object code starting at address 400H. Type

X400 RETURN

and a "7" will appear in the bottom right hand corner of your monitor screen.

The X400 means 'execute the program starting at address 400H'. An H is not necessary after the 400 because the Editor assumes that all addresses following *commands* are in hex. However, as previously stated, it assumes that all numbers in the *program* (source code) are decimal unless told otherwise.

Unfortunately the screen still has all the other information on it, because we did not take any steps to clear it. We will shortly extend our program to do this, but first let us find out how to use the Monitor to inspect those parts of memory where our program resides.

### Using The Monitor

To enter the Monitor from EDASM (where you are now) type

X RETURN

To examine our assembled program (remember it has been assembled to start at 400H) type

E 400 RETURN

which means 'Examine address 400H'. Leave a space between E and 400. Also note that the Monitor assumes that all addresses are in hex, so an H is not necessary.

Your screen should now look like this:

```

MicroBee Monitor v1.0
>E 400
0400 3E >
03F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0400 3E 37 32 FF F3 C9 00 00 00 00 00 00 00 00 00
0410 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0420 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0430 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0440 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

Each pair of numbers/letters indicates the byte residing in one address, in hex form. The address at the left of each line contains the first byte in that line. That is, address 0400 (hex understood) contains 3E. Read across the line for succeeding addresses. The right hand end of this line refers to address 040FH.

You will quickly see that the addresses and bytes correspond to the assembled program which we saw previously. Don't worry if other addresses contain bytes other than 00. They will have been left over from previous programs.

It is instructive to compare the length of the source code (that is, the number of bytes) with that of the object code. In this case the source code contains 35 bytes (excluding the ORG and END statements) whereas the object code for the same program contains only 6.

While we are in the Monitor, let us use the Modify command. Suppose we change the byte in 0401H from 37 to 38. The instruction then becomes 3E38, which corresponds to LD A,38H. The program would then place an "8" on the screen instead of "7". To make the change, first move the cursor which is now pointing to the byte in 0400H (the inverse ← to the right of 3E) by typing

CTRL. S (together) Cursor moves to right and points to byte 0401H.

M For Modify  
38 Modify this byte to 38

Then type

ESC Return to Monitor command mode

G 400 RETURN Go to 400H to execute the new program

You will now have an "8" displayed in the bottom right hand corner of the screen. So we have used the Monitor, not only to Examine memory addresses, but also to Modify one of them. (Of course the *source code* has not been changed, so if we re-assemble the program and re-execute it we will be back to a "7" on the screen).

Now try using the Monitor to examine the source code. To do this, type

E 1000 RETURN (Leave a space between E and 1000)

and check what you see against the list in Fig 4-1.

To return to the EDASM, type

ESC To Monitor command mode

X RETURN If 'Memory Size?' appears, press

RETURN again

To list your source code again type

P RETURN

As with the Editor, we have only used a few basic Monitor instructions, but having seen how these are used you will be able to experiment with the others listed in the Editor-Assembler manual.

### Improving The Program

Now let us get back to our original program and add some refinements.

You will recall that the original BASIC program which we proposed had a CLS instruction, to clear all the miscellaneous information from the screen before placing the "7" on it. How do we do this in assembly language? Remember the screen is, in effect, 1024 memory addresses, many of which contain numbers which are the ASCII code for the character in the corresponding screen position. So we have to replace those numbers with the ASCII code for Space (SP) which is 32, in all 1024 addresses. (We won't bother to determine which positions contain characters). This sounds like a formidable task, but it is not nearly as bad as it seems. Here is the program:

```

LD      BC,1024
LD      HL,0F000H
NXTLOC LD  (HL),32
INC      HL
DEC      BC
LD      A,B
OR      C
JP      NZ,NXTLOC

```

We will examine the instructions one by one.

```
LD BC,1024
```

The BC register pair is used as a counter, and is loaded initially with the number of operations required. Note that a register *pair* is used because the number to be loaded is greater than 255. The BC register pair will be decremented after each operation and tested to see if it has reached zero. If not, another operation will be performed.

```
LD HL,0F000H
```

The address of the first position of the screen is loaded into the HL register pair. Here again, we must use a register *pair* because addresses are two-byte numbers.

```
NXTLOC LD (HL),32.
```

This instruction is identified by a *Label* (NXTLOC). We mentioned labels previously, when we were discussing the layout of the source code. Remember, we TAB'd across before we typed the instruction? Well, here is our first label, and when typing this line, type the label immediately after the line number. You will see why we need a label in a minute when we discuss the operation of the program.

The instruction LD (HL),32 loads 32 into the address identified by the HL register pair. The number in HL is F000H, so 32 is placed in address F000H. So the first screen position now contains a space.

```
INC HL
```

The HL register pair is then incremented, so that it now contains F001H, the address of the *next* screen position.

```
DEC BC
```

The BC register pair is decremented, so it now contains 1023.

```
LD A,B
```

```
OR C
```

These two instructions set the Zero flag if the BC register pair contains zero. We need to test if the DEC BC instruction resulted in a zero, indicating that we had finished all 1024 positions. Unfortunately the DEC BC instruction does not affect the Z flag, so we have to resort to a devious method of doing this. If the BC register pair contains zero (0000H), both the B and C registers contain 00H. If we load B into A, A will then contain 00H. If we then OR C (with A) the result will be 00H, and the Z flag will be set. Note that this will *only* occur if both B and C contain 00H.

```
JP NZ,NXTLOC
```

The first time through the program the Z flag will not be set because BC contains 1023. So the program jumps (or 'loops') back to the instruction labelled NXTLOC, and address F001H is loaded with 32.

The process continues until the DEC BC instruction results in zero, by which time all screen addresses will have been loaded with 32 and the screen will consequently be cleared.

We can now combine this program with the number 7 program, so that we clear the screen before we place the "7" in position.

Do not forget to place an ORG statement at the start and an END statement at the end. You can probably type this without help, but, for the last time, here are the detailed typing instructions.

First, are you in EDASM? If not, remember: to enter EDASM from BASIC, type

```
EDASM RETURN (Memory Size?)
```

```
4000 RETURN for a 16K Microbee
```

```
or 8000 RETURN for a 32K Microbee
```

To enter EDASM from Monitor, type

```
ESC
```

```
X RETURN
```

Now you are in EDASM. To clear your old program, type

```
Z RETURN
```

To enter your improved program, type

```
I RETURN
```

```
TAB ORG TAB 400H RETURN
```

```
TAB LD TAB BC,1024 RETURN
```

```
TAB LD TAB HL,0F000H RETURN
```

```
NXTLOC TAB LD TAB (HL),32 RETURN
```

```
TAB INC TAB HL RETURN
```

```
TAB DEC TAB BC RETURN
```

```
TAB LD TAB A,B RETURN
```

```
TAB OR TAB C RETURN
```

```
TAB JP TAB NZ,NXTLOC RETURN
```

```
TAB LD TAB A,55 RETURN
```

```
TAB LD TAB (0F3FFH),A RETURN
```

```
TAB RET RETURN
```

```
TAB END RETURN
```

```
BREAK
```

Now assemble the program by typing

```
A/WE RETURN
```

Your screen should now look like this except that two top lines will have scrolled out of sight:

```

0400      00100      ORG      400H
0400 010004      00110      LD      BC, 1024
0403 2100F0      00120      LD      HL, 0F000H
0406 3620      00130 NXTLOC LD      (HL), 32
0408 23      00140      INC      HL
0409 0B      00150      DEC      BC
040A 78      00160      LD      A, B
040B B1      00170      OR      C
040C C20604      00180      JP      NZ, NXTLOC
040F 3E37      00190      LD      A, 55
0411 32FFF3      00200      LD      (0F3FFH), A
0414 C9      00210      RET
0000      00220      END
000000 Total errors

```

NXTLOC 0406

Note that the address of the label NXTLOC has been calculated by the assembler as 0406, and is listed at the bottom of the assembled program in what is called a Symbol Table. Note also that the JP NZ, NXTLOC instruction has been assembled as C20604, the 0604 being the address of NXTLOC *with the bytes reversed*.

Now run the program by typing

X400 RETURN

and you will be rewarded with a clear screen, and a 7 in the lower right hand corner. The Editor prompt (\*) and flashing cursor will also be on the screen, indicating that you have RETURNed to the Editor command mode. If you wish, you may list your program again by typing

P RETURN

### Saving And Loading Source And Object Code

Source code may be saved on tape from the EDASM by typing

S "NAME" RETURN

The NAME may be up to six characters long. Don't forget to start your tape recorder on RECORD before pressing RETURN!

To load previously saved source code first enter EDASM, then type

G "NAME" RETURN

OR G\* RETURN

G stands for Get. The second form of the command will load the *next* source code file on the tape, whatever it may be. The tape recorder, of course, must be on PLAY.

Object code must be saved via the Monitor. To save object code residing from RAM address 0400H to 0500H, first enter the Monitor then type

W "NAME" M 0400 0500 RETURN

W stands for Write and M indicates a Machine language file.

If you want the program to execute immediately (when it is subsequently loaded), type

W "NAME" M 0400 0500 0400 RETURN

To load previously saved object code type

R "NAME" RETURN

or R RETURN

(R stands for Read)

The second form of the command will load the next file on the tape. You can also load object code directly from BASIC by typing

LOAD "NAME" RETURN

or LOAD RETURN

### Moving Around The Microbee

Here is a summary of the instructions which will enable you to move around between BASIC, EDASM and Monitor.

TO GO FROM	TO	TYPE
EDASM	MONITOR	X RETURN - M
MONITOR	EDASM	ESC X RETURN
EDASM	BASIC	X 8021 RETURN - 0E-Y
BASIC	EDASM	EDASM RETURN
MONITOR	BASIC	ESC G 8021 RETURN
BASIC	MONITOR	USR (49155) RETURN <Reset> <M>

8021H is BASIC warm start address.

C000H (49152 decimal) is EDASM entry point.

C003H (49155 decimal) is MONITOR entry point. USR

# your computer

MAGAZINE FOR BUSINESS AND PLEASURE

\$2.50

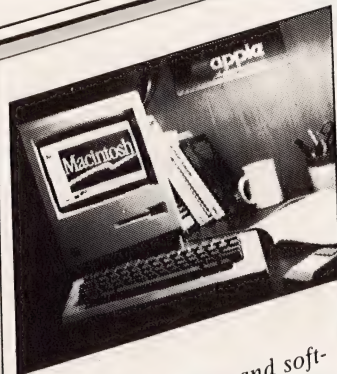
Incorporating "Your Business Computer"

Into computing? This is your magazine! Just look what's in it for you!

## NEWS



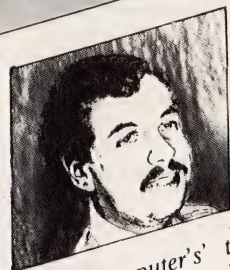
'Your Computer' brings you all that's interesting, innovative and inventive in the microcomputing world — news of products, plans and politics to keep you up-to-date with what's going on in this fast-moving industry.



The latest machines and software from all the computer manufacturers are reviewed each month in 'Your Computer'. Keep up-to-date with what's available, and use our reviews to help you assess which products are most useful for you.

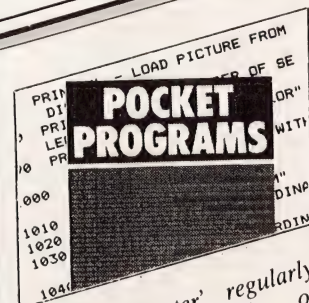
## REVIEWS

## TUTORIALS



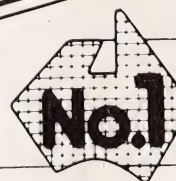
Les Bell

'Your Computer's' tutorials include regular series on such popular subjects as the BASIC programming language and dBase II, probably the biggest-selling database program of them all. Written by such well-known industry experts as Les Bell, they're an invaluable aid to learning how to make computers work for you.

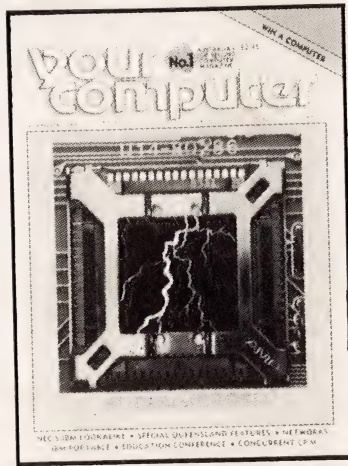


'Your Computer' regularly publishes all kinds of programs written both by professional programmers and readers, and ranging from games to business uses, utilities to additions and alterations to well-known programs. All kinds of computer users and enthusiasts will find programs to suit them in our pages!

## PROGRAMS



AUSTRALIA'S  
TOP-SELLING  
COMPUTER  
MAGAZINE  
AUDITED SALES



## Your Computer

is a magazine for all computer users and enthusiasts — there's something for everyone in our pages. We cover games and utility programs for hobbyists, and every second month we publish 'Your Business Computer' as a separate section for business computer users. 'Your Computer' offers topical features on all aspects of the computing world, written in understandable everyday language, along with reviews of new products to help you find your way round the fast-growing market, and regular columns giving specific advice on all the most popular brands of microcomputer. If computers play any part in your life at all, don't miss your monthly issue of 'Your Computer' — without it, you just won't know what's going on!

Available at your newsagent now!

# ASSEMBLER INSTRUCTIONS

## Labels and Pseudo Operators

This chapter discusses instructions which we write for the Assembler, as distinct from those we write for the CPU, which we looked at in Chapter 3. These instructions never reach the CPU, but they are used by the Assembler while generating the object code.

### ASCII Calculation

In the last chapter, to place a 7 on the screen, we first loaded the ASCII code for 7 into the A register, as follows:

```
LD A,55
```

This instruction can also be written

```
LD A,'7'
```

and the Assembler will automatically convert the 7 into its ASCII equivalent. This will occur for any numbers or letters enclosed by apostrophes.

### Labels

We saw how labels can be used in the last chapter. We need them while writing the program, because we do not know at that stage what address the labelled instruction will occupy in the final object code. The Assembler works this out for us, and any source code instruction which refers to a label will be converted into object code which includes the calculated address of that label.

Firstly, labels may contain up to six characters, either letters or numbers, *but must start with a letter*. They must not consist solely of any register names (A, B, DE etc), or source code mnemonics (LD, OR, etc) or flag names (NZ, NC, etc), or pseudo-ops (ORG, END, etc) (see below). The reason for this is simply so that the Assembler does not confuse the label with an instruction.

Secondly, labels must start with a letter so that the assembler can differentiate between labels and numbers. For the same reason, numbers must start with a digit, and that is why hexadecimal numbers starting with a letter must have a 0 preceding them.

### Pseudo Operators

You have already met the first two 'Pseudo-Ops' in the last chapter:

```
ORG nn    Start the assembled object code at address nn
END       End of source code
```

The following instructions mostly tell the assembler to load specific bytes into the object code, or reserve addresses for future use. Their use will be clarified by sample programs in Chapters 6 and 7.

VZ 200 (IN)

DEFB n or DB n Define Byte

Assemble the byte defined by the source code directly into the object code. For example,

```
DB 0FFH
```

will result in this object code:

```
0410 ??
0411 FF
0412 ??
```

(assuming 0411 is the next address available during the assembly process)

VZ 200 (IN)

DEFW nn or DW nn Define Word

Similar to DB, except that two bytes are defined. For example

```
DW 0DDFFH
```

will result in this object code:

```
0410 ??
0411 FF
0412 DD
0413 ??
```

Note that the low order byte is placed in the low address, and the high order byte in the high address, as with load instructions (ie in reverse order).

NOT IN VZ 200

DEFM 'ABCD ...' Define Message

Assemble the bytes which are the ASCII codes for the characters enclosed by apostrophes into the object code. For example,

```
DEFM 'ABCD'
```

will result in this object code:

```
0410 ??
0411 41
0412 42
0413 43
0414 44
0415 ??
```

VZ 200 (IN)

EQU nn Equate

Permanently equate a label with a two-byte number. For example, if we wish to refer to the start of screen RAM a number of times during a program, we may write,

```
SCREEN EQU 0F000H
```

Then, the instruction

```
LD HL,SCREEN
```

will load HL with 0F000H

NOT IN VZ 200

DEFL nn Define Label

Temporarily equate a label with a two-byte number. However, a new DEFL instruction will assign a new value to the label. For example,

```
SCREEN DEFL 0F000H
LD HL,SCREEN

SCREEN DEFL 0F200H
LD HL,SCREEN
```

The first LD HL,SCREEN instruction loads HL with 0F000H (as the EQU example), but the second LD HL,SCREEN instruction loads HL with 0F200H.

VZ 200 IN

DEFS n or DS n Define Space

Reserve n addresses for use during the program. For example,

```
POSN DS 2
```

will reserve two addresses, the first one identified by the label POSN. These may be used to hold temporary information during execution of the program; for instance, the current screen position of an 'alien invader'. The information could be stored by

```
LD (POSN),HL
```

and retrieved by

```
LD HL,(POSN)
```

Likewise, the instruction

```
FLAG1 DS 1
```

can be used to hold any of 255 different conditions. As above, the condition can be stored by

```
LD (FLAG1),A
```

and retrieved by

```
LD A,(FLAG1)
```

IMPORTANT NOTE:

The DB, DW, ~~DEFM~~, DS instructions must be placed at the end of the program. Otherwise the bytes contained in the allocated addresses will 'look like' object code to the CPU and it will try to execute them. (The only way they can be placed inside a program is if a JP or JR instruction is used to 'bypass' the CPU around them.)

Miscellaneous

NOT IN VZ 200

DEFR 16 Define Radix 16 (ie hexadecimal)

Regard all numbers as hex. This instruction may be over-ridden by a D after the number

+ n or minus n.

Add or subtract specified number to number in instruction. For example,

```
SCREEN EQU 0F000H
LD HL,SCREEN + 20H
```

will load HL with F020H

As another example,

```
LD A,'Z' + 1
```

will load A with 91

As another example,

```
FLAG DEFS 2
LD A,(FLAG + 1)
```

will reserve two addresses which we can use independently. The first can be accessed, as usual, by (FLAG), while the second can be accessed as above.

; Ignore remainder of this line. For example,

```
SLA D; MPLY BY TWO
```

This allows you to put comments in your source code program, like the REM statement in BASIC.

### What ORG To Use

In the Microbee, an area of RAM starting at 400H has been set aside for object code of machine language programs. This should be your normal ORiGin (in fact, if you omit the ORG statement the Assembler will default to ORG 400H).

If you have a very long program the object code will run past 1000H, where the source code starts, and will over-write it, thus destroying it. However, for this length of object code the source code would have to be approximately 8K, which represents quite an involved program. If this is likely to be a problem, there are a number of solutions:

- Ignore destruction of the source file (assuming you have saved it).
- Set aside an area at the top of memory by misleading the Editor when responding to the 'Memory Size?' question. For example, in a 32K Microbee, answer 6000 instead of 8000. This leaves 2000H addresses which you can use for object code by using the instruction ORG 6000H.
- Start the source code at somewhere other than 1000H. This can be done by using the Editor's Z instruction. For instance Z2000.

There is an area of RAM above screen RAM from F400H to F7FFH which may be used for storage of object code (ORG 0F400H).

Note that if you are using an 80 x 24 screen format (possible on some Microbees) the screen occupies up to F77FH so you should ORG 0F780H.

There is a significant advantage in using ORG F400H if your object code will not extend beyond F7FFH, because this area of RAM is not destroyed by a cold start, (other areas can be). Consequently, if you have battery back up you can leave utility programs here permanently.

# PROGRAMMING TECHNIQUES

*Messages, arithmetic, number conversions, tables, sorting, inputting and displaying strings and numbers.*

Chapter 4 described and showed how to write and run a program to clear the screen and place a number on it. In this chapter we will progress from this simple beginning using increasingly more sophisticated programs and subroutines. The operation of each will be discussed in detail, so that you can learn some of the more common programming techniques used. You will also find the subroutines useful for incorporating into your own programs, which shortly you will be writing.

It must be noted here that the programs as written are only one way of achieving the desired result, and that there are almost as many ways to write a program as there are programmers. Moreover, the same program could be written in different ways for different reasons: to occupy as little RAM space as possible, or for maximum execution speed, or, as for the programs in this book, for ease of reading and understanding the source code. These objectives are not necessarily in conflict; generally the best programs are readable, do use minimum memory and run fast.

### Messages

From Chapter 4 we know how to place an ASCII character on the screen, so it is a simple expansion of this to write a message. Let us write MISSION SUCCESSFUL on the bottom line, starting at screen address F3D6H

```
LD DE,0F3D6H ;FIRST SCREEN POSITION
LD A,'M' ;ASCII OF M
LD (DE),A
INC DE ;NEXT SCREEN POSITION
LD A,'I'
LD (DE),A
etc
```

This will achieve the desired result, but is rather tedious, so let us look at a slightly better way.

```
(ORG 400H)
LD DE,0F3D6H
LD HL,TEXT
LD BC,18
LDIR
(RET)
```

```
TEXT DB 'M'
      DB 'I'
      DB 'S'
      DB 'S'
```

```

DB 'I'
DB 'O'
DB 'N'
DB 32
DB 'S'
etc
(END)

```

As before, DE is loaded with the address of the first screen position for our message. HL is loaded with a label TEXT which, when assembled will be the address of the first of a number of Defined Bytes. We met the DB instruction briefly in Chapter 5. Each Defined Byte is the ASCII code for one letter of the message. BC is used as a counter, and is loaded with the number of letters in the message (not forgetting the space). The LDIR instruction first loads address F3D6H with the contents of address TEXT (i.e., ASCII of M), then increments both DE and HL and decrements BC. Now DE contains F3D7H, and HL contains the address of the next Defined Byte. The contents of that address (ASCII of I) is then loaded into F3D7H. The DE and HL registers are again incremented and the BC register decremented, and the whole process continues until BC = 0. By this time the whole message has been loaded into the required place on the screen, exactly as with the previous program, but using a more efficient source code.

If you decide to assemble and run this program, don't forget to include the ORG, RET and END instructions shown in brackets as we did in Chapter 4. In this case the RET instructions must be placed *before the DB statements* to prevent the CPU trying to execute the defined bytes as if they were instructions!

An even better way to write this program is to use the DEFM instruction:

```

(ORG 400H)
LD DE,0F3D6H
LD HL,TEXT
LD BC,18
LDIR
(RET)

TEXT DEFM 'MISSION SUCCESSFUL'
(END)

```

USE \*

The Assembler automatically loads the ASCII code of each character of the message into successive addresses starting at address TEXT. The operation of the program is exactly the same as before.

### Multiplication and Division

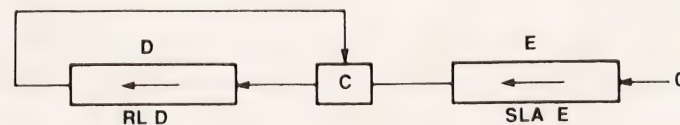
We mentioned multiplication briefly in Chapter 1, noting that if we move a binary number one place to the left (by adding 0 to the right hand side) we multiply it by 2. In the case of a single register this can be achieved with the SLA instruction. For a register pair, for instance DE, we must use two instructions:

```

SLA E
RL D

```

The operation is illustrated below:



\* LDIR - INC DE & HL  
DEC BC. & LOADS DE ADDRESS WITH CONTENTS OF HL

For multiplication by 4,8,16 etc we simply repeat the process the required number of times.

Unfortunately we sometimes need to multiply by other numbers, particularly ten. One way to multiply by ten is to first multiply by two, then by eight, and add the two results. For example, for a single register, supposing the number to be multiplied is in A:

```

SLA A ;MPLY BY 2
LD C,A ;SAVE
SLA A ;MPLY BY 4
SLA A ;MPLY BY 8
ADD A,C ;MPLY BY 10

```

A similar routine can be used for the HL register pair but is rather long because each multiplication by two requires two instructions, SLA L and RL H. A better routine is:

```

LD D,H ;SAVE HL
LD E,L ;IN DE
ADD HL,HL ;MPLY BY 2
ADD HL,HL ;MPLY BY 4
ADD HL,DE ;MPLY BY 5
ADD HL,HL ;MPLY BY 10

```

The first two lines load the number into DE, saving it for future use. The next line adds the number to itself, effectively multiplying by two. The next line repeats the process, thus multiplying the original number by four. The original number is then added from DE, resulting in a multiplication by five. The last line results in a total multiplication by ten. Note that this routine can only be used with HL, IX and IY as BC and DE cannot be added to themselves.

A very simple way of multiplying by any number is by successive addition. Suppose we want to multiply the number in HL by 54:

```

PUSH HL ;SAVE HL
POP DE ;IN DE
LD B,53
LOOP ADD HL,DE
DJNZ LOOP ;HLx54

```

The number, as before, is saved in DE, although this time we have used a different method. B is used as a counter and the number is successively added to itself the required number of times. The DJNZ instruction decrements the B register, tests for zero, and if not zero jumps to LOOP to repeat the process. Note that the first time the DJNZ instruction is encountered we have already multiplied by two, so B must be loaded with one less than the required multiplier. This would be a very long way of doing it if we used pencil and paper, but the CPU is very fast and, in fact, the above routine would only take 645 microseconds! (This figure is based on a Microbee running at 2 MHz: for one running at 3.375 MHz the time would be proportionately less). The lesson here is that a very simple program will perform multiplications by quite large numbers with insignificant delays.

Note that we are only discussing multiplication of *integers* with a result which can be held in a register pair, i.e., 65535 max. This will probably satisfy your requirements for some time, but Chapter 9 introduces the concept of multiple precision arithmetic for larger integers, and discusses representation of decimal numbers.

Division is most easily achieved by successive subtraction. For instance, to divide the number in HL by the number in DE, with the answer in BC:

```

LD      BC,0
OR      A      ;C FLAG = 0

LOOP    SBC     HL,DE
        INC     BC
        JP      NC,LOOP
        ADD     HL,DE
        DEC     BC

```

The OR A instruction resets the C Flag to 0 in preparation for the SBC (Subtract with Carry) instruction following. (There is no Subtract instruction on register pairs without Carry). The divisor is then successively subtracted from the dividend, incrementing the counter, BC, each time. When the result is negative (C Flag is set) the loop stops. We now have to reverse the last subtraction by adding DE to HL so that the remainder is left in HL. We have also counted one more than we should have, so we decrement BC which will then contain the quotient.

#### Clearing the Screen

We used a Clear Screen routine in Chapter 4 to demonstrate some basic ideas. Now that we have met and used the LDIR instruction, here is a more efficient routine for this purpose:

```

LD      HL,0F000H
LD      DE,0F001H
LD      BC,1023
LD      (HL),32
LDIR

```

See if you can discover how it works for yourself.

#### Converting ASCII to Decimal Digits

It is frequently necessary to convert the ASCII code for a decimal digit (1 to 9) to the digit it represents. If you look at Appendix 2 you will find that the ASCII code for

```

0 is 48
1 is 49
2 is 50
—
—
9 is 57

```

So, to do the required conversion, it is simply necessary to subtract 48. Suppose we had entered a digit from the keyboard and this resulted in the ASCII code of that digit being placed in the A register. We can then

```
SUB 48
```

which will subtract 48 from A, leaving the number required in A. There is another slightly more efficient way of doing this which uses one byte of object code instead of two:

```
AND 0FH
```

Let us see why this works. Referring to Appendix 2 again, you will find that the

```

ASCII code for 0 is 30H which is 00110000
1      31H      00110001
2      32H      00110010
—
—
9      39H      00111001

```

If we AND any of these numbers, say the ASCII for 2, with 0FH we get

```

32H  00110010
0FH  00001111
AND  00000010

```

leaving us with 2. Because the first (most significant) four bits of 0FH are zeroes, the first four bits of the ASCII code are 'masked out', leaving the required digit in the A register. This way of using the logical AND instruction is known as *masking*.

#### Converting Decimal Digits to ASCII

Reversing the above procedures, to convert a digit in the A register to its ASCII equivalent, leaving the answer in the A register, we can use:

```
ADD A,48
```

(Note that for the ADD instruction we have to specify the A register).

We can also use:

```
OR 30H
```

See if you can understand for yourself why this works.

#### Hex-Binary Conversion

Let us now suppose that we have a one-byte number displayed in hex form on the screen. We will write a program to calculate the number represented by the hex symbols and store it. For example, suppose the first two screen positions display '4B'. Address F000H therefore contains 52 (ASCII of 4) and F001H contains 66 (ASCII of B). The value 4B hex is to be calculated and loaded into the A register.

What we have to do is convert each ASCII code into the actual number it represents. We have seen how to do this with digits from 1 to 9, but with hex symbols A to F we have a problem. Look at Appendix 2 again:

```

SYMBOL ASCII
0      48
1      49
2      50
—      —
—      —
9      57
BREAK
A(10)  65
B(11)  66
—      —
—      —
F(15)  70

```

You will see that we cannot simply subtract 48 from ASCII A for example, to get the right answer (10) because there is a break in the code between 9 and A. (The code was invented long before its use for hex representation was ever envisaged). So our program will have to detect whether the symbol is a digit or a letter, and then make an appropriate correction to the subtraction. Having done this it will have to multiply the resulting number in the left hand position by 16, and then add this to the number in the right hand position. In our example, 4B hex =  $4 \times 16 + 11 = 75$ . Here is the program:

```

LD      A,(0F000H)  ;LEFT HAND SYMBOL
CALL    CALC
SLA     A           ;MULTIPLY BY 2
SLA     A
SLA     A
SLA     A           ;MULTIPLIED BY 16
LD      L,A         ;STORE
LD      A,(0F001H)  ;RIGHT HAND SYMBOL
CALL    CALC
ADD     A,L         ;RESULT IN A

CALC    CP          65      ;LETTER OR NUMBER?
        JP          NC,LETTER
        SUB         48      ;NUMBER-SUBTRACT 48
        RET
LETTER  SUB         55      ;LETTER-SUBTRACT 55
        RET

```

This is the first time we have used a CALL to a subroutine, although we discussed CALLS briefly in Chapters 2 and 3. The CALL CALC instruction causes the program to jump to subroutine CALC, which detects whether the symbol is a letter or a number, and does the appropriate arithmetic. Before the jump occurs the address of the instruction after the CALL (i.e., SLA A) is PUSHed (automatically) on to the stack. When the RET instruction is encountered this address is POPed off the stack (again automatically) and the program jumps back to SLA A.

The operation of the program should be quite clear with the notes against each operation.

### Tables

Now let us write a program to do the reverse of the previous one, but instead of doing it mathematically we will use a conversion, or 'look up' table.

Let us suppose that we have a number in the A register which we want to display in hex form on the screen. We will have divide the number by 16, convert the result to its ASCII form, and this will be the left hand symbol. Then we will convert the remainder to ASCII form, and this will be the right hand symbol. For example, using the same number as in our previous example,  $75/16 = 4$  remainder 11 = 4B hex. We will do the division by successive subtraction, then jump to a subroutine which converts the resultant dividend to the ASCII value of its hex symbol. The remainder is treated likewise. The two ASCII values are loaded separately into the two registers of a register pair (DE). This is then loaded into the required screen address. Here is the program:

```

LOOP    LD          BC,0      ;ZERO B & C
        INC         C         ;COUNT
        SUB         16        ;FROM NUMBER IN A
        JP          NC,LOOP   ;JUMP IF NOT NEGATIVE
        ADD         A,16      ;RESTORE LAST SUB

```

```

DEC     C                ;CORRECT COUNTER
CALL    ASCII            ;CONVERT AND LOAD D
LD      E,D              ;SAVE IN E
LD      C,A              ;REMAINDER
CALL    ASCII            ;CONVERT AND LOAD D
LD      (0F230H),DE      ;PLACE ON SCREEN

ASCII   LD          HL,TABLE
        ADD         HL,BC
        LD          D,(HL)
        RET

TABLE   DB          48      ;ASCII 0
        DB          49
        DB          50
        DB          51
        DB          52
        DB          53
        DB          54
        DB          55
        DB          56
        DB          57
        DB          65      ;ASCII A
        DB          66
        DB          67
        DB          68
        DB          69
        DB          70

```

A few points in the above program should be noted. We have zeroed the BC register pair and then used only the C register to contain our numbers, knowing that the largest number would be 15. Consequently B remains at zero. The reason we used the BC pair was so that we could add HL,BC as there is no instruction ADD HL,C. The subroutine ASCII takes the number in BC, converts it to ASCII, and loads the result into D. The table works by adding the number to the address of the start of the table. The contents of each address is the Defined Byte we have placed in it which is the ASCII of the number concerned. The LD D, (HL) instruction puts this into D. One other point should be noted. We placed the left hand ASCII digit into E and the right hand one into D. When the instruction LD (0F230H), DE is executed the numbers are reversed, E going to F230H and D going to F231H.

Note that a table of the type used above is the only way to generate a number which has no mathematical relationship to the original number. There are many instances where such a lookup table is required.

To shorten the source code of the above program the DB's can be listed as DW's (Defined Words). Each DW consists of two DB's so the list is only half as long. Remember that the DW will be loaded into memory low byte first, so we must reverse the bytes in each DW statement. For this purpose it is much easier to list them in hex, thus avoiding calculation into decimal form. The above list would then be:

```

TABLE DW 3130H
        DW 3332H
        DW 3534H
        DW 3736H
        DW 3938H
        DW 4241H
        DW 4443H
        DW 4645H

```

The assembled object code would be identical to the previous one.

**Sorting**

Suppose we have a list of numbers sitting in memory, starting at address LIST. Let us write a program to sort them in ascending order. To do this we will use the special indexing feature of the IX register. We will assume they are two-byte numbers, so they will be stored as follows:

ADDRESS	CONTENTS
LIST	LOW BYTE, FIRST NO.
LIST + 1	HIGH BYTE, FIRST NO.
LIST + 2	LOW BYTE, SECOND NO.
LIST + 3	HIGH BYTE, SECOND NO.
etc.	

To sort them we will use a 'bubble sort' which works by running down the list and comparing each number with the next. If the next number is smaller than the previous one, the two are swapped. If not, no swap occurs. This process continues down the list, and then is repeated, starting from the beginning of the list again. The list is scanned repeatedly in this manner until no swap has occurred during the scan. Here is the program, assuming the list contains ten numbers.

```

SORT      LD      B,10          ;NO. OF NUMBERS
          XOR     A              ;A = 0
          LD      IX,LIST       ;LOW BYTE FIRST NO.
SORT 1    LD      L,(IX)        ;HIGH BYTE FIRST NO.
          LD      H,(IX+1)      ;LOW BYTE NEXT NO.
          LD      E,(IX+2)      ;HIGH BYTE NEXT NO.
          LD      D,(IX+3)      ;C FLAG = 0(A = A)
          OR      A             ;COMPARE BY SUB
          SBC     HL,DE          ;IF NEGATIVE
          JP      C,NOSWAP      ;IF ZERO
          JP      Z,NOSWAP      ;RESTORE
          ADD     HL,DE          ;SWAP
          LD      (IX),E
          LD      (IX+1),D
          LD      (IX+2),L
          LD      (IX+3),H
          LD      A,1           ;SET SWAP FLAG
          INC     IX
          INC     IX            ;NEXT NUMBER
          DJNZ    SORT 1
          CP      1             ;SWAP?
          JP      Z,SORT        ;REPEAT IF SWAP
          JP      Z,SORT

```

Note that the indexed addresses IX + 1, IX + 2 etc can be accessed directly without having to add the index first to the IX register. The A register is used as a flag to indicate whether a swap has occurred during the scan. The B register, as usual, is used as a counter so that the efficient DJNZ instruction can be used.

**Input**

All the programs that we have discussed so far have involved manipulation of data within the computer. We have assumed that we had numbers already in various registers. We now come to the important question of how to input information from the keyboard during the execution of a program. In BASIC we simply type

```

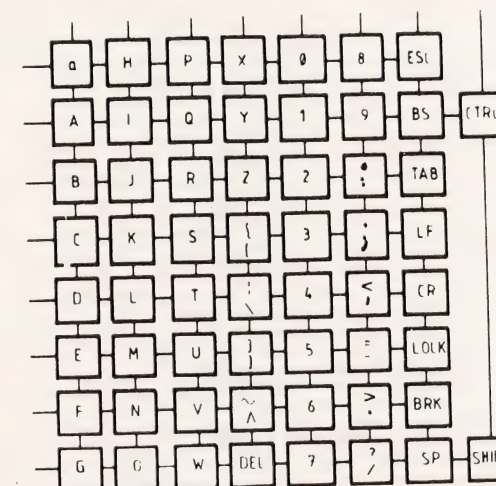
INPUT A
or
A0$ = KEY$

```

and key in the data. How do we do this in machine language? To answer this question we will first look at how the keyboard works.

The keyboard is laid out electrically in a matrix, as shown in Fig 6-1, consisting of eight columns and eight rows.

Fig. 6-1



Each key is located at the junction of a column and a row. For instance, the A key is at the junction of column 1 and row 2. When this key is pressed column 1 is connected to row 2. Each key, therefore, has its own unique combination of column and row which, when the key is pressed, are connected. The task the computer has is to find out when any column and row are connected. If they are it can work out which key has been pressed.

To do this, if the computer has been told to expect an input from the keyboard, it 'scans' the keyboard continuously looking for columns and rows which are connected. It checks the first row with each column in turn, then the second row with each column, and so on. If it does not find a connection it repeats the whole process. Each complete scan of the keyboard takes only a fraction of a second so there is no chance of it missing a key that has been pressed.

How do we instruct the CPU to scan the keyboard? We could write a long and complicated routine to do this, but this has already been written for the BASIC Interpreter, and we are going to make use of it. If you refer back to the memory map in Chapter 2 you will see that the BASIC interpreter ROM occupies addresses 8000H to BFFFH. The 'scan keyboard' routine starts at 8006H, so we will simply

```
CALL 8006H
```

The CPU will now continually scan the keyboard as described above, waiting for a key to be pressed. When it finds one, it loads the A register with the ASCII code of that key, and then RETURNS to the main program. We can now take the byte from the A register for use in our program.

**For Owners of 64K Microbees**

64K Microbees do not have a ROM based BASIC interpreter. To use BASIC subroutines like 8006H it is necessary to first load Microworld Basic (TBASIC) from disk, which will then occupy the same locations as the ROM based interpreters.

**Inputting Strings**

Let us use what we have just learned to write a program to input a string (perhaps

the name of a player) from the keyboard, display it on the screen, and store it for future use somewhere in RAM. We will include a Back Space routine for sloppy typists, and limit the number of characters to 20. The RETURN key will complete the input. Here is the program:

```

(ORG 400H)
LD IX,PLAYER ; STORAGE ADDRESS
LD HL,0F167H ; SCREEN POSITION
LD B,20 ; CHR COUNTER
NXTCHR CALL 8006H ; INPUT KEY
CP 8 ; BACK SPACE?
JP Z,BACKSP
CP 13 ; CARRIAGE RETURN?
JP Z,FINISH
INC B
DEC B
JP Z,NXTCHR
LD (IX),A ; TO STORAGE
LD (HL),A ; TO SCREEN
INC IX ; NEXT STORAGE ADDRESS
INC HL ; NEXT SCREEN POSITION
DEC B ; COUNT CHARACTERS
JP NXTCHR

BACKSP DEC IX
DEC HL
INC B
LD (IX),32 ; SPACE
LD (HL),32
JP NXTCHR

FINISH LD (IX),32 ; STORE SPACES
INC IX
DJNZ FINISH
(RET)

PLAYER DEFS 20
(END)

```

After initialising the IX, HL and B registers, the CALL 8006H instruction causes the CPU to scan the keyboard continuously until a key is pressed. When this occurs the ASCII code for the key is loaded into the A register. The next instruction, CP 8 compares the A register with 8 by subtracting 8 from it without actually changing it. The numeral 8 is the ASCII code for Back Space, so if the Back Space key has been pressed the CP 8 instruction sets the Z flag. Consequently the next instruction jumps the CPU to BACKSP which decrements both address pointers and blanks out the previous character. The CP 13 instruction checks for Return which indicates the end of the string. If all 20 positions have been used B will have been decremented to 0. To test for this the INC B, DEC B instructions will set the Z flag if this is the case, and jump the CPU back to NXTCHR. Only Back Space or Return are now permitted.

Normal entries (i.e. B > 0) are loaded into the screen address and consequently displayed, and also stored in one of 20 RAM addresses reserved by the DEFS 20 instruction, starting from the address identified by PLAYER. The screen address and storage address are both incremented and the counter decremented before returning to NXTCHR. The three instructions at FINISH, load spaces into the unused RAM addresses.

If you assemble and execute this program by itself, don't forget, as before, to use ORG and END statements for the Assembler and a RET instruction for the CPU — before the DEFS statement as indicated.

### Displaying Strings

Having stored a string in memory as in the previous program, it is a simple matter to recall it and display it at any desired screen position, perhaps to show a player's name with his score:

```

LD DE,0F220H
LD BC,20
LD HL,PLAYER
LDIR

```

We have used the same number of addresses which we originally reserved in RAM, but this time we are using BC as the counter instead of B so that we can use the LDIR instruction. You can now see why we loaded spaces into the unused addresses; if we had not done so we would display any 'garbage' that happened to be there.

### Inputting Numbers

We have just seen how to input strings from the keyboard, display them, store them, and re-display them. We will now look at the same processes for numbers.

With numbers the process of *displaying* the number entered from the keyboard is no different to strings: we simply load the ASCII code for each digit into screen RAM. However, we need to *store* the number as a number, not as a series of digits. To do this we have to convert each digit from its ASCII code to its actual value, and then multiply by its positional value in the decimal system. For example, if we key in

1234

the computer stores

31H  
32H  
33H  
34H

We must convert these to

1  
2  
3  
4

and multiply by the positional value of each, and add the results:

1x1000  
plus 2x100  
plus 3x10  
plus 4x1

The following program will do what we have just described for any number up to 65535, loading the result into HL.

```

      (ORG 400H)
      LD HL,0
      LD IY,0F0F0H ;SCREEN POSITION
      LD B,0
READ  CALL 8006H
      CP 8
      JP Z,BACKSP ;BACK SPACE?
      CP 13
      JP Z,CALCNO ;CARRIAGE RETURN?
      LD (IY),A
      INC IY
      INC B
      JP READ ;NO. OF DIGITS

BACKSP DEC IY
      LD (IY),32 ;SPACE
      JP READ

CALCNO DEC IY
      LD DE,1 ;UNITS
      CALL DECML
      LD DE,10 ;TENS
      CALL NZ,DECML
      LD DE,100 ;HUNDREDS
      CALL NZ,DECML
      LD DE,1000 ;THOUSANDS
      CALL NZ,DECML
      LD DE,10000 ;TEN THOUSANDS
      CALL NZ,DECML
      (RET)

DECML LD A,(IY)
      AND 0FH ;ASCII TO DIGIT
LOOP  JP Z,DONE
      ADD HL,DE ;MPLY
      DEC A
      JP LOOP
DONE  DEC IY
      DEC B
      RET
      (END)

```

HL is to contain the number, so it is initialised at 0. B counts the digits as they are entered and displayed on the screen. We do not know how many digits the number will contain until after the Return key is pressed. When this occurs we calculate the value of the number by working backwards from the Units end. The DECML subroutine takes the ASCII of the digit from the screen, converts it to its actual value, multiplies it by the value of DE by successive addition, and loads the result into HL. For the Units digit the DE value is 1, and for the next digit it is 10 and so on. Note that if a digit is 0 the Z flag is set by the AND 0FH instruction. For digits other than 0 it is set by the DEC A instruction after the appropriate number of ADDs.

When all the digits have been processed the DEC B instruction in the DECML subroutine sets the Z flag before returning to the main program. Once this is set no further calls to the subroutine occur and the program is finished. The HL register will contain the cumulative total of all the calculations, which is the required number.

If you want to test the operation of this program you will need a way of seeing what number is actually in HL. To do this you may add the instructions

```

      LD DE,0F23AH
      CALL 0DCB8H

```

to the end of the main program. The routine this calls at DCB8H is in the Monitor ROM and displays the number in HL at screen position DE (in hex).

### Displaying Numbers

Displaying a stored number on the screen in decimal form is really the reverse of what we have just done. We have to divide the number successively by 10000, 1000, 100, 10, 1 to determine each digit. We then convert each digit to its ASCII equivalent and load these into successive screen positions. This time we have no way of knowing how many digits the number will have, and this presents a problem. If the number was 123, unless we take special measures it would display as 00123. This, of course, is quite correct, but looks messy, so the program below has provision to prevent this occurring.

```

      (ORG 400H)
      LD DE,0F230H ;SCREEN POSITION
      XOR A ;A=0, C FLAG=0
      LD (PRINT0),A
      LD BC,10000
      CALL DECML
      LD BC,1000
      CALL DECML
      LD BC,100
      CALL DECML
      LD BC,10
      CALL DECML
      LD BC,1
      CALL DECML
      (RET)

DECML SBC HL,BC
      JP C,DISPLAY
      INC A
      LD (PRINT0),A ;PRINT ZEROES
      JP DECML

DISPLAY ADD HL,BC ;RESTORE
      EX AF,AF'
      LD A,(PRINT0)
      CP 0 ;PRINT ZEROES?
      JP Z,SPACE
      EX AF,AF'
      ADD A,30H ;DIGIT TO ASCII
      LD (DE),A ;DISPLAY
      INC DE
SPACE  INC DE
      XOR A ;A=0, C FLAG=0
      RET

PRINT0 DEFS 1 ;FLAG
      (END)

```

The XOR A instruction resets both the A register and the C flag to 0. This value is loaded into a reserved RAM address labelled PRINT0 which is a flag to determine

if zeroes should be printed. Each digit is calculated in the DECML subroutine by successive subtraction of 10000, 1000 etc using A as a counter. When A is incremented for the first time the PRINT0 flag is also incremented. When the count for each digit is finished the program jumps to DSPLAY. Unfortunately we need the A register to test the PRINT0 flag, so we use the alternative AF register pair for this purpose. If the flag is still zero the program jumps to SPACE and nothing will be displayed on the screen. If the flag has been incremented the AF registers are exchanged again to retrieve the digit stored in the A register. This is converted to ASCII and displayed at the current DE screen position. Note that if A = 0 a 0 will be displayed if any previous digit has been other than 0.

### Miscellaneous Program Routines

#### JP AND JR INSTRUCTIONS

The Jump Relative (JR) instruction was mentioned briefly in Chapter 3. It is an instruction to the CPU to jump a certain number of addresses, forwards or backwards, from its present position. The number of addresses that can be jumped is limited to 127 forwards or 128 backwards (counting from the *end* of the instruction itself). The following JR instructions can be used in place of the equivalent JP instructions:

```
JR LABEL
JR C,LABEL
JR NC,LABEL
JR Z,LABEL
JR NZ,LABEL
```

The Assembler will calculate the number of addresses to be jumped for you and insert this into the object code. As an example, the assembled object code for the CALC subroutine from the Hex-Binary conversion program (page 56) is shown below with the alternative jump instructions. The JP NC, LETTER instruction assembles as D20804, 0408 being the address of LETTER. The JR NC, LETTER instruction assembles as 3003, there being three addresses between the end of the instruction and LETTER. Note that you save one byte this way, but remember the limitation on the number of addresses that can be jumped.

ADDR	CODE	LINE	LABEL	MNEM	OPERAND
0400		00100		ORG	400H
0400	FE41	00110	CALC	CP	65
0402	D20804	00120		JP	NC, LETTER
0405	D630	00130		SUB	48
0407	C9	00140		RET	
0408	D637	00150	LETTER	SUB	55
040A	C9	00160		RET	
0000		00170		END	
00000 Total errors					
LETTER 0408      CALC      0400					

ADDR	CODE	LINE	LABEL	MNEM	OPERAND
0400		00100		ORG	400H
0400	FE41	00110	CALC	CP	65
0402	3003	00120		JR	NC, LETTER
0404	D630	00130		SUB	48
0406	C9	00140		RET	
0407	D637	00150	LETTER	SUB	55
0409	C9	00160		RET	
0000		00170		END	
00000 Total errors					

LETTER 0407      CALC      0400

### Rules for Subroutines

Here are some important guidelines for your subroutines:

1. As we mention in Chapter 2 the Stack is used for storing both PUSHed register pairs and also the return address for a subroutine. For this reason all stack operations within a subroutine must be matched; that is, you must have the same number of PUSHes and POPs — so that the return address is ready to be POPed at the end of the subroutine.
2. Often you will want to use register pairs in a subroutine which contain information from the main program which you do not want to lose. In this event PUSH all such register pairs at the start of the subroutine and POP them in reverse order at the end.
3. Place all subroutines at the end of the main program, otherwise you will have to jump over them.



## microbee Series III now with Built-in Silicon Disk . . .

The best selling **microbee** is the new 32K Communicator Series III and it is no wonder. The Series III has now been expanded to include 32K of battery backed memory (programs are saved when the power is switched off), 40K of software built-in ROM including Microworld Basic, Wordbee wordprocessor, Telcom 3 with real time clock/alarm, machine code monitor, self testing and full communications capability. Additional 32K of software on ROM can be added to the built-in ROM board to make the Series III 32K **microbee** the only low cost personal computer on the market with a built-in silicon disk.

The **microbee** is widely used in Homes, Schools and Business in Australia, New Zealand and worldwide. The features that have made it popular are the functions such as programmable serial and parallel ports, sound effects, high resolution PCG graphics with 512 by 256 bit resolution, programmable 80 by 24 or 64 by 16 screen character display, terminal emulation and communications capability.

The new Telcom ROM which has been added to the Series III turns your **microbee** into a real time clock with alarm features. It has full communications capability so your Microbee can become a home terminal and talk to other computers across the world via the telephone lines using the BEEMODEM. The new Telcom now has built-in testing, machine code monitor and even a calculator!

**microbee** has a large range of proven peripherals such as a fast low cost dot matrix printer, a new low cost daisy wheel printer and the new high quality 8401 XY plotter. A large range of software and other accessories means that you will always be able to use your **microbee** to meet virtually any need.

**microbee 32K Communicator Workstation. . . \$599.00**  
includes 32K **microbee** Series III and Hi-res Green Monitor.

Available separately:

**microbee 32K Series III . . \$499.00**  
**Hi-res.Green Monitor . . . \$149.50**  
**Hi-res.Amber Monitor . . . \$169.50**  
**Swivel Base . . . . . \$24.50**

**bankcard**  
welcome here

**DIRECT ORDERS**  
**PHONE (02)487 2711**  
**TELEX AA72767**  
**APPLIED TECHNOLOGY**  
**RETAIL PTY LTD**



# microbee®

Designed and manufactured in Australia

### microbee technology centres

N.S.W.  
1 Pattison Ave,  
Waitara 2077, N.S.W.  
Phone (02) 487 2711

VIC.  
729 Glenferrie Rd,  
Hawthorn 3122, Vic.  
Phone (03) 819 5288

W.A.  
141 Stirling Highway,  
Nedlands, W.A.  
Phone (09) 386 8289

S.A.  
151 Unley Road,  
Unley 5061, S.A.  
Phone (08) 272 1384

QLD.  
455 Logan Road,  
Stones Corner, Qld 4120  
Phone (07) 394 3688

**FACTORY** Koala Crescent, West Gosford 2250.  
Phone (043) 24 2711

### Authorised Dealers in:

•Coffs Harbour  
(066)52 7530

•Lismore  
(066)21 8180

•Melbourne  
(03)233 9665

•Townsville  
(077)75 4000

•Cairns  
(070)51 4341

•Milton  
(07)371 6888

•Port Lincoln  
(086)82 2666

•Mount Tom Price  
(091)89 1383

•Launceston  
(003)31 7890

•Wauchope  
(065)85 3311

# GRAPHICS AND SOUND

*Programmable character generator, LORES graphics, HIRES graphics, moving graphics, and sound effects.*

One of the major advantages of programming in machine language rather than BASIC is the speed of execution.

This is particularly important with 'real time' graphics programs like the many video games which are so popular. You may have tried your hand at programming one of these games in BASIC and found that it was disappointingly slow. In machine language you will find that the equivalent program needs time delays built in to slow it down, just so you can see what is happening on the screen!

### The Character Generator

The Character Generator generates all the characters which appear on the screen. The characters are numbered from 0 to 255. For characters 0 to 127 the character number is the same as the ASCII code for that character, and cannot be changed. For example, in BASIC

```
PRINT CHR(72)
```

will always print H on the screen. On the other hand, characters 128 to 255 are *programmable*, and can be made into Inverse or Underline ASCII characters, or LORES dots, or shapes of your own design. If they have not been programmed (i.e., the default state) they will be Inverse ASCII characters, the character number being obtained by adding 128 to the ASCII code. For example,  $128 + 72 = 200$ , so that

```
PRINT CHR(200)
```

will print an Inverse H. The programmable part of the Character Generator is referred to as the PCG.

As you know, you can place an ASCII character on the screen in machine language by

```
LD A,72  
LD (0F000H),A
```

which will place H in the first screen position. Similarly, the programmable character numbers 128 to 255 can be used. As above,

```
LD A,200  
LD (0F000H),A
```

will place an inverse H in the first screen position (assuming the characters have not been programmed).

The entire character set can be printed with the following program:

```

ORG 400H
LD HL,61440 ;FIRST SCREEN POS.
LD B,0 ;CHARACTER 0
LD A,255 ;COUNTER
LOOP LD (HL),B ;DISPLAY
INC HL ;NEXT POSITION
INC B ;NEXT CHARACTER
SUB 1 ;DECREMENT COUNTER
JP NC,LOOP
RET
END

```

This will print all the 'non printable' characters (control characters with codes 0 to 31) which do not respond to a BASIC PRINT CHR() instruction, as well as the printable characters.

(It is interesting to do the same thing with the equivalent BASIC program

```

FOR A = 0 to 255
POKE 61440 + A,A
NEXT A

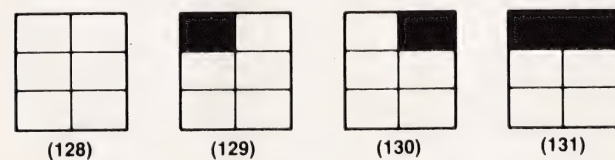
```

and compare the speed of execution).

We will now look at the various ways we can program characters 128-255 from machine language.

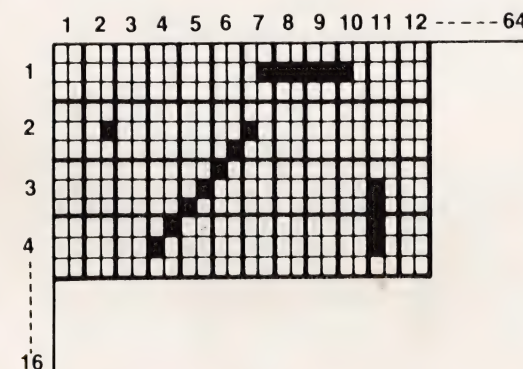
### LORES GRAPHICS

In the BASIC LORES mode the programmable characters are used to provide chunky graphics dots by creating characters consisting of six blocks as shown:

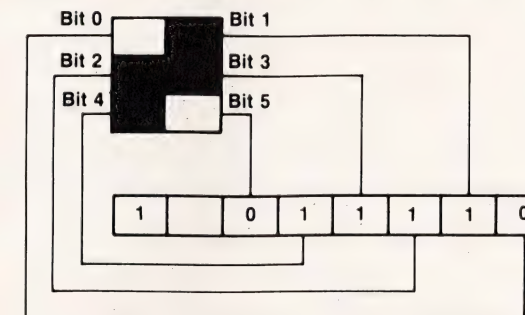


Any or all of the blocks may be 'filled', each filled block then becoming a LORES dot. Any of the characters may be placed in any screen position so that any pattern of LORES dots can be produced. See Fig 7-1. Because the characters are two dots wide and three dots high, the total number of dot positions on the screen is 2x64 horizontally and 3x16 vertically, because there are 64 screen positions in each of 16 lines. Thus the LORES graphics resolution is 128x48.

Fig. 7-1



The character numbers are coded as follows:



Each of the six blocks is allocated one bit of the character number byte, as shown. If a dot is set its corresponding bit will be set (= 1). If it is reset its corresponding bit will be reset (= 0). Because the numbers must be in the range 128-255, Bit 7 must always be 1. Bit 6 may be either 1 or 0. Consequently the character number for the pattern above is either

10011110 which is 158  
or 11011110 which is 222

A short cut method of determining LORES character numbers is as follows:

128 +	1	2
OR 192 +	4	8
	16	32

For the pattern above, add  $128 + 2 + 4 + 8 + 16 = 158$   
or  $192 + 2 + 4 + 8 + 16 = 222$

To initiate LORES from machine language we use the Lores routine in the BASIC interpreter by

```
CALL 8027H
```

This immediately sets the programmable characters as described above. We can now create chunky graphics patterns by placing the appropriate LORES character into the required position on the screen. As a simple example, here is a program to place a vertical line down the centre of the screen

```

ORG 400H
CALL 8027H ;INITIALISE LORES
LD HL,0F020H ;CENTRE TOP LINE
LD DE,64 ;LINE INCREMENT
LD B,16 ;LINE COUNTER
LOOP LD (HL),149
ADD HL,DE
DJNZ LOOP
RET
END

```

149 = 128 + 1 + 4 + 16

Character 149 is placed in the same position on successive lines by incrementing the screen position by 64 until all lines have been done.

**Using the Set Routine**

The BASIC SET x,y command will set a LORES dot at screen co-ordinates x,y where x is in the range 0-127 and y is in the range 0-47. It does this by calculating what LORES character is required in which screen position, taking account of dots already set.

To use BASIC'S inbuilt SET routine from your own machine language program, the x and y co-ordinates must first be loaded into the HL and DE register pairs respectively. The routine is located at 8030H, so it may be accessed with a CALL to this location. Let us use this routine to set two LORES dots on the right centre of the screen:

```

ORG 400H
CALL 8027H ;INITIALISE LORES
LD HL,100 ;X = 100
LD DE,23 ;Y = 23
CALL 8030H ;SET DOT 100,23
INC HL ;X = 101
CALL 8030H ;SET DOT 101,23
RET
END

```

If you tried this program you will have found that the second dot was not set. The reason for this is that the 8030H routine uses the HL and DE register pairs without saving them and consequently the original values which they contained (100 and 23) are lost. Therefore, in accordance with our Rules for Subroutines (Chapter 6) we must PUSH the two register pairs containing the information we wish to retain before going to the subroutine, and POP them in reverse order after returning. (It is more usual to do the PUSHing and POPing in the subroutine itself, but in this case we do not have access to it). Our program will then look like this:

```

ORG 400H
CALL 8027H
LD HL,100 ;X=100
LD DE,23 ;Y=23
PUSH HL ;SAVE X
PUSH DE ;SAVE Y
CALL 8030H ;SET DOT 100,23
POP DE ;RECOVER Y
POP HL ;RECOVER X
INC HL ;X=101
CALL 8030H ;SET DOT 101,23
RET
END

```

You will find that the second dot will now be set. Now let us draw a line using this routine:

```

ORG 400H
CALL 8027H
LD HL,100
LD DE,23
LD B,10 ;COUNTER
LOOP PUSH HL
PUSH DE
CALL 8030H
POP DE

```

```

POP HL
INC HL ;X=100 TO 110
DJNZ LOOP
RET
END

```

This program will plot a line from 100,23 to 110,23 without the need for us to determine LORES characters and screen positions.

Here are some other routines which can be CALLED in the same way as the set dot routine. Remember, you must first initialise LORES with CALL 8027H, and then load HL with the x co-ordinate and DE with the y co-ordinate.

```

8030H Set dot (as above)
8033H Reset dot (clear)
8036H Invert dot
8039H Test for dot set:Z not set, NZ set

```

**Using the Plot Routine**

The programs which we have suggested so far for drawing LORES lines are fine for vertical and horizontal lines, but require a lot of fiddling to draw oblique lines. Just as we can access the inbuilt BASIC SET routine as above, so we can use the BASIC PLOT routine, which will plot a line in LORES dots between two points which we will call x1,y1 and x2,y2.

As before, to use this routine we must first initialise LORES with CALL 8027H. We must then load the co-ordinates x2,y2,x1,y1 into RAM addresses 0F9H, 0FBH, 0FDH and 0FFH respectively. That is,

```

x2 into 0F9H
y2 into 0FBH
x1 into 0FDH
y1 into 0FFH

```

Now we must indicate the type of line we require by loading address 0E8H with either

```

20H to Set dots (normal)
52H to Reset (ASCII of R)
49H to Invert dots (ASCII of I)
48H to Invert Y axis (PLOT)(ASCII of H)

```

We can now CALL the PLOT routine at 803CH.

Here is a sample program to plot a LORES line from 20,10 to 107,37.

```

ORG 400H
CALL 8027H
LD HL,107
LD (0F9H),HL ;X2=107
LD HL,37
LD (0FBH),HL ;Y2=37
LD HL,20
LD (0FDH),HL ;X1=20
LD HL,10
LD (0FFH),HL ;Y1=10
LD A,20H
LD (0E8H),A ;SET DOTS
CALL 803CH ;PLOT LINE
RET
END

```

**HIRES Graphics**

The LORES Set and Plot routines discussed above can also be used for HIRES graphics simply by changing the CALL to LORES (8027H) to the corresponding HIRES command,

```
CALL 8024H
```

The range of co-ordinates will then be 0-511 for x and 0-255 for y.

However, there is a difficulty. The HIRES program uses a part of RAM (200H to 3FFH) which is normally used by the machine language stack and EDASM 'scratch'. (This term means storage of information essential to the proper running of the program). Consequently, when HIRES is called all this information is lost and the stack cannot operate correctly.

To overcome this problem we will have to move the entire scratch area somewhere else before HIRES is called, and return it to its original position at the end of the program. We can do this with two LDIR instructions, but we must first decide where to put it. If we ORG at 600H instead of 400H we will be able to fit it in from 400H to 5FFH, so this is what we will do.

We also have to set up another stack for use by the program. However, before we do this we must store the current top of stack address somewhere because we will need it later. We will place it at the top of memory in addresses 3FFEh and 3FFFh (for a 16K Microbee) with the instruction

```
LD (3FFEh),SP
```

Now we can set up the new stack immediately below these addresses with

```
LD SP,3FFDH
```

All stack operations within the program will now take place from 3FFDH down.

At the end of the program, before the RET instruction, we must replace the stack in its original position by

```
LD SP,(3FFEh)
```

This is because the top of the stack (in its original position) contains the return address to the EDASM program.

Here is a HIRES Plot routine demonstrating these techniques.

```
ORG 600H
LD (3FFEh),SP ;STORE STACK POINTER
LD HL,200H
LD DE,400H
LD BC,200H
LDIR ;MOVE EDASM SCRATCH
LD SP,3FFDH ;MOVE STACK
CALL 8024H ;INITIALISE HIRES
LD HL,500
LD (0F9H),HL ;X2=500
LD HL,250
LD (0FBH),HL ;Y2=250
LD HL,50
LD (0FDH),HL ;X1=50
LD HL,20
LD (0FFH),HL ;Y1=20
LD A,20H
LD (0E8H),A ;SET DOTS
```

```
CALL 803CH ;PLOT LINE
LD HL,400H
LD DE,200H
LD BC,200H
LDIR ;MOVE SCRATCH BACK
LD SP,(3FFEh) ;RETRIEVE STACK POINTER
RET
END
```

**Designing Characters**

As we mentioned at the start of this chapter the PCG characters numbered 128 to 255 can be designed by you to whatever pattern you like, so that you can make your own alien invader, or frog hopper or whatever.

You may have learned to do this in BASIC, in which case the following will be familiar. The difference as far as machine language is concerned will be in how we move these shapes around the screen, which will be much faster and which we will learn about shortly.

Every character which appears on your screen, not only the programmable ones (128-255) but also the non-programmable ones (0-127) is made up of a matrix of dots, eight across and sixteen down. Character 72, H, for instance, is made up as shown in Fig 7-2.

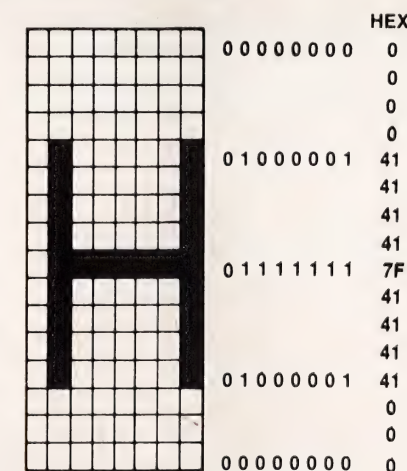


Fig. 7-2

Each horizontal line of eight dots is represented by eight bits, 1's for dots which are set and 0's for dots which are not set. Consequently each of the sixteen horizontal lines is represented by one byte. In Fig 7-2 the top four lines have all dots not set, so each line is represented by 00000000 or 00H. The next four lines are 01000001, or 41H, the next is 01111111, or 7FH and so on. Figure 7-2 should make this quite clear.

So every character is defined by sixteen bytes, and these bytes must be stored somewhere in memory. Each character number has its own 16 memory addresses which contain the data for that particular character. Appendix 5 lists the addresses of the top line of all the characters. For instance, the top line of character 193 has address 64528. The following fifteen addresses correspond to the other fifteen lines of this character. To create our own character 193, therefore, we must load these addresses with the necessary bytes for the pattern we require. Having done this we can place this pattern anywhere on the screen in exactly the same way we use for the ASCII characters 0-127. That is,

```
LD A,193
LD (0F220H),A
```

which will place our own character 193 near the centre of the screen.

Let us try an example. Suppose we want to make character 193 the pattern shown in Fig 7-3.

Fig. 7-3

			HEX
0001	1000	1	8
0011	1100	3	C
0011	1100	3	C
0001	1001	1	9
0001	1001	1	9
0001	1010	1	A
0011	1100	3	C
0101	1000	5	8
1001	1000	9	8
1001	1000	9	8
0001	1000	1	8
0001	0100	1	4
0011	0110	3	6
0010	0011	2	3
0010	0001	2	1
0010	0001	2	1

Write out the binary number for each horizontal line and look up the hex equivalent in Appendix 1. (It is quicker to do this by dividing the character down the middle and looking up each nibble independently). Write the hex byte beside each line. Now load these bytes into the sixteen addresses for character 193 with the following program:

```

(ORG 400H)
LD HL,PCGDTA
LD DE,64528 ;CHR 193 TOP ADDRESS
LD BC,16
LDIR
(R)
PCGDTA DW 3C18H
        DW 193CH
        DW 1A19H
        DW 583CH
        DW 9898H
        DW 1418H
        DW 2336H
        DW 2121H
(END)

```

Note that, as before (Chapter 6), we have used DW statements instead of DB's and that each pair of bytes in the DW statement is in reverse order. Remember also that the DW statements must be at the end of the program.

If we want to design more than one character we can do so with virtually the same routine as above as long as the characters are consecutive. For instance, to design characters 193 and 194 all we have to do is to change LD BC,16 to LD BC,32 and add the extra PCGDTA information.

### Calculating Character Data Addresses

Sometimes it is necessary for a program to be able to calculate the data addresses for any particular character. If you look at Appendix 5 again you will see that the address for the top line of character 0 is F000H or 61440. The address for the top line of each succeeding character is obtained by adding 16. Consequently the

address for the top line of any character can be obtained by multiplying the character number by 16 and adding the result to F000H, i.e.,

$$\text{Address} = \text{CHR} \times 16 + \text{F000H}$$

Just a moment! Addresses F000H to F3FFH are screen addresses aren't they? Quite correct, that is normally so. However, the addresses from F000H to F7FFH, which include the screen addresses, can be changed so that they become the data addresses for characters 0-127. The data is stored in the character generator ROM. To change from screen RAM to character generator ROM a software instruction is used. Data for characters 0-127 can then be read. We will see exactly how to do this in Chapter 8 when we write a graphics screen dump program. This dual memory allocation was shown in Fig 2-2, Chapter 2, but we did not elaborate on it at that time.

### Moving Graphics

Now that we know how to design our own characters we will want to start moving them around the screen if we are to develop our own games, or for other programs. Some of them we will want to move continuously, perhaps the alien invaders, while others, the 'goodies' will move only when certain keys are pressed.

To move a character from left to right on one line of the screen we must place it in successive positions on the line. However, if we simply did this we would fill the line with the character, so we must delete the character from the preceding position as we move across, like this:

```

MOVE LD HL,0F2C0H ;LEFT HAND 12TH LINE
      LD (HL),193 ;ASSUME PROGRAMMED
      LD (HL),32 ;SPACE
      LD A,L ;END OF LINE CHECK
      AND 63
      CP 63 ;END OF LINE?
      INC HL
      JP NZ,MOVE

```

To check if we have reached the end of the line we make use of the fact that the last address of every line has bits 0,1,2,3,4,5 all set. Check this for yourself with Appendices 6 and 1. (This happens because there are 64 addresses per line, and  $64 = 2^6$ , so the binary bit pattern of the last 6 bits is repeated after each cycle of 64). To test for this condition we load the low byte of the screen position into A and AND it with 00111111. This 'masks out' bits 6 and 7 of the address. We now Compare the result of the AND operation with 00111111. A zero result sets the Z flag and indicates the end of the line. Note that the Z flag is not affected by the INC HL instruction, so the Jump to MOVE will occur repeatedly until the end of the line is reached.

If you run the above program you won't see a thing! That is because it all happens so fast that it is completely invisible. Consequently we must insert a delay in the program which leaves the character in each screen position for a definite time, long enough to see it. This leads us to the subject of delays.

### Delays

There are many ways to insert time delays into a program, but they all operate on the basic principle of giving the CPU some useless task to do which takes some time. The easiest way is simply to count down to zero from a number loaded into the B or BC register. Let us look at the times involved.

```

LD      B,255
DELAY DEC B          ;2 MICROSECONDS
JP      NZ,DELAY      ;5 MICROSECONDS

```

Total time  $7 \times 255 = 1785$  microseconds. This is the longest delay we can obtain using a single register and is much too short for graphics programs. Consequently we will have to use a register pair, like this:

```

LD      BC,65535
DELAY DEC BC          ;3 MICROSECONDS
LD      A,B           ;2
OR      C             ;2 BC = 0?
JP      NZ,DELAY      ;5

```

Total time is  $12 \times 65535$  microseconds, which is approximately 0.8 seconds. This time applies to Microbees running at 2 MHz. For Microbees running at 3.375 MHz the time will be reduced to approximately 0.5 seconds.

If you want a really long delay you can use nested loops, like this:

```

LD      B,255
DELAY1 LD HL,65535
DELAY2 DEC HL
LD      A,H
OR      L
JP      NZ,DELAY2
DJNZ   DELAY1

```

This will give you three minutes and twenty seconds at 2 MHz!

### Back to the Program

After that diversion let us now put a delay into our moving character program so we can see what is happening.

```

ORG      400H
LD      HL,0F2C0H
MOVE LD   (HL),193
CALL    DELAY      ;LEAVE A WHILE
LD      (HL),32
LD      A,L
AND     63
CP      63
INC     HL
JP      NZ,MOVE
RET

DELAY LD   BC,10000
LOOP DEC  BC
LD      A,B
OR      C
JP      NZ,LOOP
RET
END

```

If you run this program you will see character 193 move across the screen on the 12th line at a reasonable pace. If you have not programmed the character it will be inverse A. You can, of course, vary the speed by changing the 10000 loaded into BC.

To move left instead of right we must DEC HL instead of INC HL. To test for the start of the line (left hand position), after masking out bits 6 and 7 of the address (AND 63) we Compare with 00000000 (CP 0).

To move down we must add 64 to HL. If HL then exceeds the end of screen RAM, F3FFH, H will contain F4H. We can test for this by Comparing H with F4H. There is a subtle difference here to our previous technique. In this case we have to increment HL *before* we test because the change to H does not occur until after the bottom of the screen has been passed.

```

ORG      400H
LD      HL,0F020H      ;START POSITION
MOVEDN LD   DE,64
LD      (HL),193
CALL    DELAY
LD      (HL),32
ADD     HL,DE
LD      A,H
CP      0F4H
JP      NZ,MOVEDN
RET

DELAY LD   BC,10000
LOOP DEC  BC
LD      A,B
OR      C
JP      NZ,LOOP
RET
END

```

To move up we subtract 64 from HL and Compare H with EFH to test whether HL is below screen RAM.

### Changing the Moving Character

Often you will want to change the character as it moves across the screen to give the impression of moving arms or legs, for instance. Suppose we do this to our character 193, Fig 7-3. We must, of course, design another character with the legs in a different position, and this will become character 194 if we load the PCG data for both characters consecutively as described previously. Having done that, all we have to do is load characters 193 and 194 alternately into successive screen positions. We can do that like this, using our move left to right routine:

```

ORG      400H
LD      HL,0F2C0H      ;START POSITION
MOVE LD   A,193
BIT     0,L             ;ODD OR EVEN?
JP      Z,LEGS1
LD      A,194
LEGS1 LD   (HL),A
CALL    DELAY
LD      (HL),32

```

```

LD      A,L
AND     63
CP      63
INC     HL
JP      NZ,MOVE
RET

DELAY   LD      BC,10000
LOOP    DEC     BC
LD      A,B
OR      C
JP      NZ,LOOP
RET
END

```

We test bit 0 of the L register which contains the low byte of the screen position. This will alternate between 0 and 1 as we increment HL. If 0, 193 is loaded into address HL, so character 193 is placed into even screen positions. If 1, 194 displaces 193 in the A register, so character 194 is placed into odd screen positions.

### Controlling Movement

Most graphics games require an input from the keyboard to control one of the characters on the screen (usually you, the defender or the good guy!). Let us now see how to do this.

We have seen in Chapter 6 that if we CALL 8006H we input the next key pressed into the A register. This is fine for arithmetic type programs, but no good for most graphics programs because the main program stops until a key is pressed. In graphics programs there are usually alien invaders or other things moving all the time so this action must continue whether you are pressing keys or not. To overcome this problem we use

CALL 0A50AH

This is an undocumented BASIC routine (you won't find it listed anywhere) which will test whether a specific key is depressed. If it is, the routine sets the Z flag, and if not it resets it. In either case it returns to the main program. The following program demonstrates how this routine is used, using the MOVE and DELAY routines we have discussed previously. It will move character 193 up, down, right or left using keys Q (or q), A (or a), "." and ",". If you run this program by itself you do not need a RET instruction because it runs in a continuous loop. To terminate the program, press RESET.

```

(ORG 400H)
CALL CLS      ;CLEAR SCREEN
LD HL,0F220H  ;START POSITION

DSPLAY LD      (HL),193
CALL DELAY
LD      A,', '
CALL 0A50AH
CALL Z,MOVLFT
LD      A,'.'
CALL 0A50AH

```

```

CALL Z,MOVRGT
LD      A,'Q'
CALL 0A50AH
CALL Z,MOVUP
LD      A,'q'
CALL 0A50AH
CALL Z,MOVUP
LD      A,'A'
CALL 0A50AH
CALL Z,MOVDN
LD      A,'a'
CALL 0A50AH
CALL Z,MOVDN
JP      DSPLAY

```

```

MOVLFT LD      (HL),32
LD      A,L      ;EDGE?
AND     63
CP      0
RET     Z
DEC     HL
RET

```

```

MOVRGT LD      (HL),32
LD      A,L      ;EDGE?
AND     63
CP      63
RET     Z
INC     HL
RET

```

```

MOVDN LD      DE,64
LD      (HL),32
ADD     HL,DE
LD      A,H      ;BOTTOM?
CP      0F4H
RET     NZ
OR      A          ;C FLAG=0
SBC     HL,DE      ;RESTORE
RET

```

```

MOVUP LD      DE,64
LD      (HL),32
OR      A          ;C FLAG=0
SBC     HL,DE
LD      A,H      ;TOP?
CP      0EFH
RET     NZ
ADD     HL,DE
RET

```

```

DELAY LD      BC,1000
LOOP  DEC     BC

```

```

LD      A,B
OR      C
JP      NZ,LOOP
RET

CLS     LD      HL,0F000H
LD      DE,0F001H
LD      BC,1023
LD      (HL),32
LDIR
RET

( END )

```

### Testing for Collisions

Many of the popular graphics games require a test to see whether you, or one of your missiles, has hit a target. If your position is identified by HL, as in the program above, all that is necessary is to test that the next position you are going to occupy has a space in it, like this:

```

TESCOL LD      A,(HL)
CP      32
CALL    NZ,CRASH

```

Note that this routine must be run before you occupy the position so, in the program above, it would be placed immediately before JP DISPLAY.

### Sounds and Sound Effects

The speaker in your Microbee is connected (via a transistor) to bit 6 of Port 2. We have said nothing up to now about PORTS, but all you need to understand for the present discussion is that you can place a byte on Port 2 with the instructions

```

LD      A,n
OUT     (2),A

```

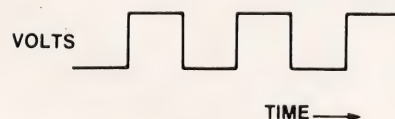
where n is a one-byte number. The OUT instruction is similar to a LD instruction in that the byte will remain on the port until displaced by another OUT instruction. Likewise, the port may be read without disturbing the data on it by

```

IN      A,(2)

```

We can make a noise in the speaker by placing 1's and 0's alternately on bit 6 of Port 2. We must leave each 1 and 0 there for a certain time which we will determine with a DELAY. The transistor between bit 6 and the speaker will apply a voltage to the speaker whenever bit 6 is set (1), and zero voltage when it is reset (0). Consequently the voltage applied to the speaker will look like this:



This is called a square wave and produces a noise from the speaker familiar to you when you

PLAY 13,8

in BASIC.

To place a 1 on bit 6 of Port 2 we must place the binary number 01000000 (which is 64) on Port 2, like this:

```

LD      A,64
OUT     (2),A

```

To place a 0 on bit 6 of Port 2 we must place 00000000 on Port 2 like this:

```

LD      A,0
OUT     (2),A

```

Let us now add a delay for both the 1's and 0's like this:

```

(ORG 400H)
NOTE LD      A,64
OUT   (2),A
CALL  DELAY
LD      A,0
OUT   (2),A
CALL  DELAY
JP      NOTE

DELAY LD      BC,95
LOOP DEC     BC
LD      A,B
OR      C
JP      NZ,LOOP
RET
( END )

```

If you run this program you will find it will give a continuous note of pitch A above Middle C if your Microbee runs at 2 MHz. If it runs at 3.375 MHz, change 95 in the DELAY subroutine to 160 to achieve the same note.

Let us see how this is calculated. Each pass of the above program (i.e., from NOTE to JP NOTE) generates one complete cycle of the musical note, represented by:



The number of complete cycles per second is called the frequency and is referred to as Hz (Hertz). The frequency of a musical note determines what our ears perceive to be its pitch, that is, whether it sounds high or low.

Suppose, as in the example above, we decided we wanted a note of pitch A above Middle C. If you look up a table of musical notes you will find that this note has a frequency of 440 Hz, or 440 cycles per second (Concert Pitch). Each complete cycle must therefore take 1/440 seconds, or 2272 microseconds. Each DELAY period must be half this, or 1136 microseconds because we use two DELAY times per cycle.



We learned in the last chapter that the DELAY routine used in the program

above takes 12 microseconds (at 2 MHz) on each pass. Therefore we need  $1136/12 = 95$  passes, which is the number we loaded into BC.

To achieve other notes on the musical scale it is necessary to understand how the scale works. Notes in each octave have twice the frequency of the corresponding note in the octave below. Each octave is divided into twelve semi-tones, each of which represents an increase in frequency by the same ratio. This ratio is 1.0595, or the twelfth root of 2. Consequently, the frequency of A# (the semi-tone above A) is  $440 \times 1.0595$  and this note can be produced by changing the initial BC value in the DELAY subroutine above in the inverse ratio. This gives  $95/1.0595 = 90$ . Likewise, to produce A one octave *higher* BC should be loaded with  $95/2 = 47.5$ , so you will have to settle for 47 or 48.

You won't normally want to continue your note indefinitely so you will have to build another delay into the program to terminate it. Below is a program which demonstrates how to do this, in which the number in DE represents the duration of the note in half cycles. The program as written will produce one note, but if a number of different notes were required, for instance to play a tune, the routine starting at NOTE could be CALLED as a subroutine for each note after loading the values for pitch and duration for that note into BC and DE.

```

      (ORG 400H)
      LD BC,50      ;PITCH
      LD DE,1000    ;TIME
NOTE  IN A,(2)      ;GET BYTE
      XOR 64        ;TOGGLE 6TH BIT
      OUT (2),A     ;BYTE OUT
      PUSH BC       ;SAVE PITCH
LOOP  DEC BC        ;DELAY
      LD A,B
      OR C
      JP NZ,LOOP
      POP BC        ;RETRIEVE PITCH
TIME  DEC DE        ;DURATION
      LD A,D
      OR E
      JP NZ,NOTE
      (RET)
      (END)

```

The program works in a slightly different way to the previous one. The three line routine

```

IN    A,(2)
XOR   64
OUT   (2),A

```

takes the byte which happens to be on Port 2, toggles the sixth bit, and places the result back on Port 2. This method has the advantage of not disturbing the other bits of the Port. This could be important under some conditions, because these bits are used for tape recorder and serial port input and output.

The pitch of the note is determined, as before, by the number loaded into BC, and the duration by the number in DE. The delay routine which determines the pitch of the note is the same as before except that it is now part of the main program instead of being a subroutine. Because the delay routine decrements BC to zero, it is necessary to PUSH BC before the routine and POP BC after it to restore it to its original value. The TIME routine determines the duration of the note and is simply a count down of DE to zero. The number in DE is actually the number of half-cycles generated.

### Sound Effects

The program above is useful when simple notes are required, but sometimes we want something more interesting. A whole range of effects can be produced by changing the pitch of the note as it sounds, from the noise of a ship's fog horn to the 'ZAP' of a laser gun! We can do this with a small addition to the previous program by incrementing or decrementing BC every so many passes of the program. In the program below the number in the L register determines how many program passes (half cycles) occur before each decrement of BC, thus controlling the rate at which the pitch changes. H is used as a counter and is re-loaded from L after each count down.

```

      (ORG 400H)
      LD BC,50      ;PITCH
      LD DE,1000    ;TIME
      LD L,40       ;PITCH CHANGE RATE
      LD H,L        ;INITIALISE COUNTER
NOTE  IN A,(2)      ;GET BYTE
      XOR 64        ;TOGGLE 6TH BIT
      OUT (2),A     ;BYTE OUT
      PUSH BC       ;SAVE PITCH
LOOP  DEC BC        ;DELAY
      LD A,B
      OR C
      JP NZ,LOOP
      POP BC        ;RETRIEVE PITCH
      DEC H         ;CHANGE PITCH?
      JP NZ,TIME    ;NO
      DEC BC        ;YES
      LD H,L        ;RESTORE COUNTER
TIME  DEC DE        ;DURATION
      LD A,D
      OR E
      JP NZ,NOTE
      (RET)
      (END)

```

You will find a surprising range of noises can be obtained with the above program just by juggling the values placed in BC, DE and L, and changing DEC BC to INC BC.

# dreamcards

## SOFTWARE



<b>CHIP-8:</b> The ultimate arcade games language (see October "Electronics Today International") -	\$49.95 ab
<b>BASIC SCREEN EDITOR:</b> At last a full screen editor! It makes programming a breeze! -	\$39.95 ab
<b>KEY SCREEN EDITOR:</b> A full screen editor for the Assembly Language Programmer. -	\$39.95 ab
<b>COMPU-B:</b> Now anyone can beat the TAB without needing to know about horse racing. -	\$39.95 b
<b>MULTIBANK:</b> The ultimate in home accounting systems at half the cost of its rivals. -	\$34.95 bc
<b>MERLIN:</b> The greatest adventure ever written for the Bee. Help Arthur find the sword. -	\$27.95 bc
<b>TREK:</b> A superb, sophisticated 32K version of this graphic/adventure (See Nov. "P.C. Games") -	\$19.95 c
<b>BARMOTH:</b> Another fine adventure set in a magical land of old. Kill the dreaded Spectre. -	\$19.95 c
<b>BUILDING OF FEAR:</b> A 4-stage adventure set in a multi-storey building. Can you escape? -	\$19.95
<b>BUSHWALKING ADVENTURE:</b> A superb adventure set in the Australian bush. Comes with a grid map. -	\$19.95 c
<b>KILLERBEES:</b> A challenging graphics/adventure style program. Save Aussie from the Bees! -	\$17.95 c
<b>SQUADRON:</b> The arcade action game of the year! (see Nov. "P.C. Games"). Like "Galaxians".	\$19.95
<b>PENETRATOR:</b> High speed arcade action. Why go to your local pinball parlour to play it? -	\$24.95
<b>ORION:</b> Stellar mines, torpedoes and lots of fun for the arcade enthusiast. -	\$17.95
<b>Z80 DISASSEMBLER:</b> An essential utility. This one generates "Labels" (unique!) -	\$17.95
<b>PSYCHOTEC:</b> The computer-psychiatrist. An amazing artificial intelligence program. -	\$17.95 b
<b>DUO 1-6:</b> Two programs per tape. Top arcade games/adventures at a budget price.	\$17.95-\$19.95

**NOTE:** An "a" after the price means that the program is supplied in EPROM only, "b" means that a detailed operating manual is included and "c" means that it won't operate on a 16K machine.

To order write to DREAMCARDS, 8 Highland Court, Eltham North 3095, Vic.  
Free Catalogues supplied on request.

Please add \$1.00 P&P per item to all orders. Sorry, we do not deal in Disc drives or disc-based Software.

# DRIVING PRINTERS

*Output port control, output to printers, ASCII screen dump and graphics screen dump.*

**A**ny reader lucky enough to own a printer will know that in BASIC, the command

LPRINT "QWERTY"

will print QWERTY on the printer. If your printer has a parallel interface you will also have to type

OUTL#1

before the LPRINT statement.

In this chapter we will learn the equivalent machine language routines to the above and illustrate them with two screen dump programs, one for ASCII characters only, and one for a mixture of ASCII and graphics characters.

## Parallel and Serial Interfaces

First of all, a word about the two types of interface, parallel and serial. When a byte of data is sent to a printer through a parallel interface the eight bits of the byte are sent at the same time through eight separate wires. The printer recognises the eight inputs as the eight bits of a byte and prints the character corresponding to the byte. Such a printer is known as a parallel printer.

When a byte of data is sent through a serial interface the eight bits of the byte are sent through a single pair of wires one after the other. To accept these the printer must also be equipped with a serial interface and is therefore known as a serial printer. The serial printer must collect the eight bits of each byte and assemble them back into a parallel form. From that point the printing operation is the same as in the parallel printer.

There must be some way of separating the bits of one byte from the bits of the next. This is done with start and stop bits and need not concern us here. The rate at which the bits are sent can be chosen at will, but the printer needs to know what the rate is, otherwise it will not be able to distinguish between the separate bits of each byte. A common standard rate for serial printers is 1200 bits per second, known as 1200 baud.

## The Microbee Output Ports

The Microbee has both a serial and a parallel port. Each port can be set up by software to be either input or output. After a cold start the serial port is set up automatically for output at 1200 baud. Consequently, if a 1200 baud serial printer is connected to it any PRINT or LPRINT commands will produce a print-out on the printer. On the other hand the parallel port is initially set up for input (for use by joysticks, external keyboards etc) and must be programmed for output if we wish to drive a parallel printer.

The BASIC OUTL#1 command mentioned above does precisely this. If we

## Microbee HACKER'S HANDBOOK

From the publishers of 'ETI' and 'Your Computer' magazines.

**Hard and soft projects for Bees of all vintages**

*For Bee owners who like to*

- Explore the innards of their machine
- Experiment with practical programming
- Put their soldering irons to use

**\$6.50  
AUST**

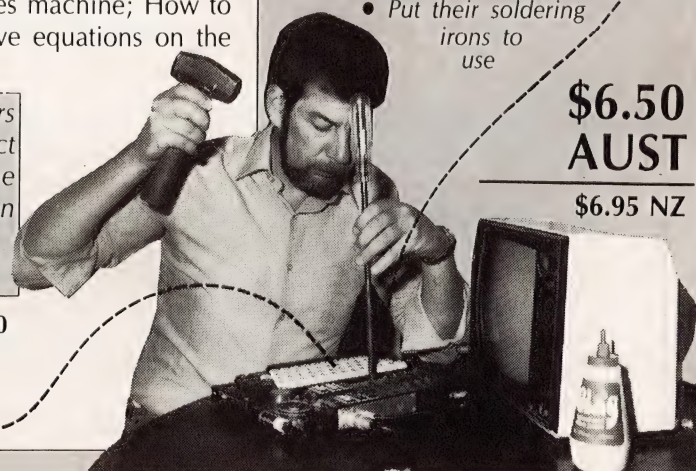
**\$6.95 NZ**

The Microbee Hackers Handbook will tell you how to get more from the Bee's graphics capabilities; How to set up for telecommunications on the Bee; How to play music on the Bee; How to build a joystick and a light pen for the Bee; How to get hard copy — cheaply; How to expand your 16K Bee to 32K; How to make the Bee a better games machine; How to turn your Bee into a facsimile decoder; How to solve equations on the Bee ... and much more.

We've also included listings of the Bee's BASIC scratch area and machine language locations — to make the Hacker's pursuits easier.

For sociable Hackers there are the contact addresses and phone numbers of all known Microbee user groups.

At your local newsagent, or from ETI Book Sales, 140 Joynton Avenue, Waterloo 2017. (Please add \$1 to the cost of the magazine, to cover postage and handling.)



have a machine language program which sends out data for printing and we have a parallel printer we must ensure that the parallel port has been set up as above. We can do this by typing OUTL#1 before entering EDASM, but the law of maximum inconvenience will ensure that you frequently forget to do this. If you go back to BASIC from EDASM you may (partially) destroy your machine language program. It is therefore desirable to include the parallel output set-up procedure in the machine language program.

### Setting the Parallel Port for Output

Here is the machine language equivalent to the BASIC OUTL#1 command which we have labelled OUTPP.

```
OUTPP LD    A,2H
      LD    (0E3H),A
      LD    A,0
      LD    (109H),A
      LD    A,0FH
      OUT   (1),A
      LD    A,80H
      OUT   (1),A
```

The first two instructions place 2H, or 00000010 into address E3H. The contents of this address determine which output devices are connected to the data to be sent. In this case, because bit 1 is set, output device #1 is connected, which is the parallel port. The next two instructions place a 0 into address 109H. This tells the CPU that it may send the first byte of data to the printer. (While the printer is reading each byte the CPU is prevented from sending the next byte until the printer is ready to receive it. Address 109H is used as a 'flag' which is set to 0 if the printer is ready, and FFH if it is not). The next two instructions send 0FH to Port 1 which actually sets the parallel port for output. The last two instructions provide the port controller with an address to which the CPU jumps when the printer is ready to receive the next byte of data. The routine at this address re-sets address 109H to 0. Don't worry if this seems complicated; you can use the routine without understanding exactly how it works.

### Sending Data to the Printer

Now we come to the heart of this chapter; how do we PRINT from machine language? Here again we will use a routine already in the BASIC ROM. The routine at 8045H will take a byte from the A register and send it to whichever output ports have been set up. To print QWERTY on your printer, therefore, you can use the following program.

```
CALL   OUTPP ;IF PARALLEL PRINTER
LD     A,'Q'
CALL   8045H
LD     A,'W'
CALL   8045H
      —
      —
      —
etc
```

If you assemble and execute this program you may find that nothing seems to happen! This is because some printers collect the bytes sent to them in an 'input buffer' but do not print anything until they receive a Carriage Return (CR) instruction, at which time they print the whole line. You may also need to tell the printer to move to the next line by sending a Line Feed (LF) instruction. The

ASCII code for CR is 13 and that for LF is 10. To get around these problems, our program now looks like this:

```
ORG    400H
CALL   OUTPP
LD     A,'Q'
CALL   8045H
LD     A,'W'
CALL   8045H
      —
      —
      —
LD     A,13
CALL   8045H
LD     A,10
CALL   8045H
RET

OUTPP LD    A,2H
      LD    (0E3H),A
      LD    A,0
      LD    (109H),A
      LD    A,0FH
      OUT   (1),A
      LD    A,80H
      OUT   (1),A
      RET

END
```

Note that if you have a serial printer you must not use the OUTPP routine as this will stop output to the serial port.

You can assemble the OUTPP routine at F400H where it will be safe from cold starts. You may then CALL 0F400H in your printer programs and thus save writing the routine into each program.

### ASCII Screen Dump

Let us now use our new knowledge to write a program to print whatever the screen is displaying out to a printer. We may want to do this at the end of a game, where the final scores are shown, or at the end of a series of calculations, where the results are listed.

What we have to do is look at each screen position in turn, determine its contents, and send that byte to the printer. After doing this 64 times we must send a CR and a LF before starting on the next screen position. (If we don't do this the printer will continue to print on the same line. After 80 characters, if it is an 80 column printer, it will generate its own internal CR and LF and the resulting print-out will have the lines displaced). After 16 lines have been scanned and printed the task is complete. Here is the program.

ADDR	CODE	LINE	LABEL	MNEM	OPERAND
F400		00100		ORG	0F400H
F400	C5	00110		PUSH	BC ;SAVE REGISTERS
F401	E9	00120		PUSH	HL
F402	CD27F4	00130		CALL	OUTPP ;OUT PARALLEL PORT
F405	0E10	00140		LD	C,16 ;LINE COUNTER
F407	0640	00150		LD	B,64 ;CHARACTER COUNTER
F409	2100F0	00160		LD	HL,0F000H ;FIRST SCREEN POSITION

```

F40C 7E      00170 LINE   LD      A,(HL)      ;GET CHR FROM SCREEN
F40D CD4580  00180      CALL  8045H          ;SEND CHR TO PRINTER
F410 23      00190      INC    HL            ;NEXT POSITION
F411 10F9    00200      DJNZ  LINE          ;END OF LINE?
F413 3E0D    00210      LD     A,0DH        ;CARRIAGE RETURN
F415 CD4580  00220      CALL  8045H          ;TO PRINTER
F418 3E0A    00230      LD     A,0AH        ;LINE FEED
F41A CD4580  00240      CALL  8045H          ;TO PRINTER
F41D 0D      00250      DEC    C
F41E 2804    00260      JR     Z,FINISH      ;LAST LINE
F420 0640    00270      LD     B,64
F422 18E8    00280      JR     LINE          ;NEXT LINE
F424 E1      00290 FINISH POP    HL
F425 C1      00300      POP    BC
F426 C9      00310      RET
              00320
F427 3E02    00330 OUTPP LD     A,2H
F429 32E300  00340      LD     (0E3H),A
F42C 3E00    00350      LD     A,0
F42E 320901  00360      LD     (109H),A
F431 3E0F    00370      LD     A,0FH
F433 D301    00380      OUT    (1),A
F435 3E80    00390      LD     A,80H
F437 D301    00400      OUT    (1),A
F439 C9      00410      RET
              00420
0000      00430      END
000000 Total errors

FINISH F424      LINE F40C      OUTPP F427

```

The program is shown assembled at F400H, and you should be able to follow its operation. To use it to obtain a print-out at the end of a BASIC program, use the instruction

P = USR (62464)

62464 being F400H in decimal. At the end of a machine language program, after returning to EDASM, use

X 0F400

If the print-out is required in the middle of a program the call to the print routine must be inserted in the program. For a BASIC program this is, as before, P = USR (62464), but for a machine language program, it will be

CALL 0F400H

Note that the above program will only print ASCII characters, that is, letters or numbers. If there are any PCG characters on the screen they will print out as something completely different, exactly what depends on the particular printer. To print out graphics characters, or a combination of ASCII and graphics characters, we have to do something completely different.

### Graphics Screen Dump

Before we start, it must first be realised that it is necessary to have a dot matrix printer with graphics capability. These printers have a print head consisting of, typically, nine pins in a vertical line. The print head passes over the paper horizontally, and at each horizontal position the required pins are hammered against the ribbon and paper to form a pattern of dots. When the printer is in normal (ASCII) mode, the patterns produced are determined by the printer, each ASCII character being formed in a matrix of nine dots vertically and eight horizontally (typically).

When the printer is in *graphics* mode, each byte sent to it actuates eight of the nine pins, bit for bit, the top pin corresponding to Bit 7 of the byte sent. (The ninth pin is not used). Succeeding bytes are printed as a vertical dot pattern in succeeding horizontal positions, left to right. Consequently, we can reproduce any pattern of dots on the screen with the same pattern of dots on paper (remember, screen characters are made up of eight dots wide by sixteen dots high). What we have to do is break down what is on the screen into its dot pattern and send this information to the printer.

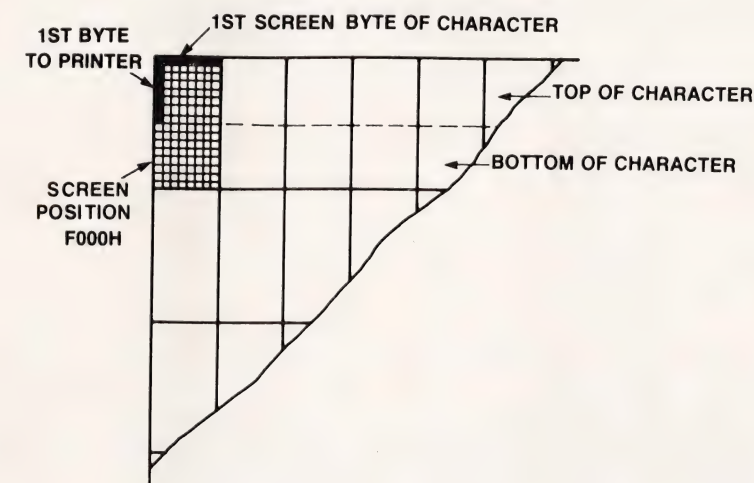


Fig. 8-1

However, this task is a little more complicated than it might at first appear to be. Let us look at the top left corner of the screen, Fig 8-1, which we know is screen position F000H. We can find out what character is in this position by looking at address F000H. If it is a PCG character (128-255) we know from Chapter 7 the address of each byte corresponding to each horizontal row of dots making up the character. Remember that the first byte is the top row, the second byte the next row down and so on. Unfortunately the printer works from left to right, not from top to bottom. We cannot just send these bytes to the printer because it would print all the characters sideways!

What we have to do is take the first bit (Bit 7) of each of the first eight bytes of the character in turn and send these as a byte to the printer. The printer will print this as the left hand column of the top half of the first character. The next byte for the printer is made up of the next bit (Bit 6) of each of the first eight bytes of the character, and so on. After eight bytes the top half of the first character has been printed, and this is followed by the top half of the next character, in address F001H, and so on for 64 screen positions. After that we must go back to address F000H and print the bottom halves of the first 64 characters. Consequently we will need to print 32 lines, alternately top and bottom of the characters, to complete the whole screen.

As indicated above we need to know the addresses of the bytes which make up each character. We already know these for PCG characters, but where are the bytes for ASCII characters 0-127? We mentioned briefly in Chapter 7 that they were in addresses F000H to F7FFH, but are not normally accessible, as some of these addresses are used for screen RAM. They are stored in ROM and can be read but not altered. To read the character ROM we must operate a switch which switches from screen RAM to character ROM. We do this by

```

LD      A,1
OUT     (11),A

```

which sends 01H to Port 11. Circuits in the Microbee do the actual switching. When we have read the information we require from the ROM we must switch

back to the screen RAM so we may look at the character in the next screen position. To switch back we

```
LD    A,0
OUT   (11),A
```

As we mentioned in Chapter 7, the address for the top line of any character is given by  $\text{CHR} \times 16 + \text{F000H}$ . See also Appendix 5.

Two other jobs have to be done. The first is to set the printer to graphics mode. Exactly how this is done will depend on the printer, but the program below demonstrates how to do this on the Microbee MB-80 printer, and a number of compatible types. To set these printers to graphics mode for a certain number of bytes we must send

```
ESC;K;CHR(x);CHR(y)
```

where the number of bytes =  $x + 256 \times y$ . In our case each line of printing will consist of  $64 \times 8 = 512$  bytes, so we will need to send

```
ESC;K;CHR(0);CHR(2)
```

at the start of each line, or codes 27,75,0 and 2, the ASCII code for ESC and K being 27 and 75 respectively.

The second job to be done is to set the line spacing of the printer so that the bottom half of each character is printed immediately under the top half, with no space in between. To do this on the above printers we must send

```
ESC;1
```

or 27,49. Here is the complete program.

ADDR	CODE	LINE	LABEL	MNEM	OPERAND
3000		00100		ORG	3000H
3000	C5	00110		PUSH	BC ;SAVE REGISTERS
3001	D5	00120		PUSH	DE
3002	E5	00130		PUSH	HL
3003	CD8830	00140		CALL	OUTPP ;OUT PARALLEL PORT
3006	2100F0	00150		LD	HL,0F000H ;FIRST SCREEN POSITION
3009	0E20	00160		LD	C,32 ;LINE COUNTER
300B	0640	00170		LD	B,64 ;CHARACTER COUNTER
300D	CD9B30	00180		CALL	LSPACE ;SET LINE SPACING
3010	CD7330	00190	LINE	CALL	PRNTGR ;SET GRAPHICS MODE
3013	7E	00200	NXTCHR	LD	A,(HL) ;GET CHR FROM SCREEN
3014	E5	00210		PUSH	HL ;SAVE SCREEN POSITION
3015	6F	00220		LD	L,A ;CHR INTO HL
3016	2600	00230		LD	H,0
3018	29	00240		ADD	HL,HL ;IMPLY BY 2
3019	29	00250		ADD	HL,HL
301A	29	00260		ADD	HL,HL
301B	29	00270		ADD	HL,HL ;IMPLY BY 16
301C	1100F0	00280		LD	DE,61440 ;OFFSET ADDR TOP ROW
301F	CB41	00290		BIT	0,C ;TEST TOP OR BOTTOM
3021	2803	00300		JR	Z,TOP
3023	1108F0	00310		LD	DE,61448 ;OFFSET ADDR 9TH ROW
3026	19	00320	TOP	ADD	HL,DE ;ADDR TOP OR 9TH ROW
3027	CD4B30	00330		CALL	PRINT ;PRINT CHR TOP OR BOT
302A	E1	00340		POP	HL
302B	23	00350		INC	HL ;NEXT SCREEN POSITION
302C	10E5	00360		DJNZ	NXTCHR
302E	3E0D	00370		LD	A,13 ;CARRIAGE RETURN
3030	CD4580	00380		CALL	8045H ;TO PRINTER
3033	3E0A	00390		LD	A,10 ;LINE FEED
3035	CD4580	00400		CALL	8045H

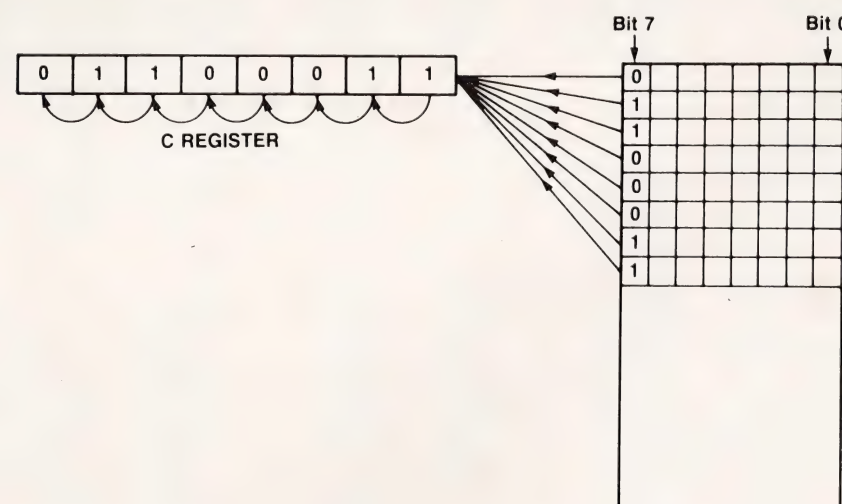
3038	CB41	00410	BIT	0,C		TEST TOP OR BOTTOM
303A	2004	00420	JR	NZ,BOT		
303C	11C0FF	00430	LD	DE,-64		
303F	19	00440	ADD	HL,DE		IF TOP START OF LINE
3040	0640	00450	LD	B,64		RE-LOAD CHR COUNTER
3042	0D	00460	DEC	C		
3043	2802	00470	JR	Z,FINISH		LAST LINE
3045	18C9	00480	JR	LINE		NEXT LINE
3047	E1	00490	POP	HL		
3048	D1	00500	POP	DE		
3049	C1	00510	POP	BC		
304A	C9	00520	RET			
		00530				
304B	3E01	00540	PRINT	LD	A,1	
304D	D30B	00550		OUT	(11),A	READ CHR ROM
304F	C5	00560		PUSH	BC	SAVE LINE COUNTER
3050	1680	00570		LD	D,128	10000000
3052	0608	00580		LD	B,8	COLUMN COUNTER
3054	C5	00590	NXTCOL	PUSH	BC	SAVE
3055	0608	00600		LD	B,8	ROW COUNTER
3057	E5	00610		PUSH	HL	SAVE TOP ADDRESS
3058	CB21	00620	NXTROW	SLA	C	SHIFT AND ZERO BIT 0
305A	7E	00630		LD	A,(HL)	
305B	A2	00640		AND	D	READ BIT
305C	2802	00650		JR	Z,BITOFF	
305E	CB01	00660		SET	0,C	IF BIT SET
3060	23	00670	BITOFF	INC	HL	NEXT ROW
3061	10F5	00680		DJNZ	NXTROW	
3063	79	00690		LD	A,C	
3064	CD4580	00700		CALL	8045H	BYTE TO PRINTER
3067	E1	00710		POP	HL	RECOVER TOP ADDRESS
3068	C1	00720		POP	BC	RECOVER COL COUNTER
3069	CB3A	00730		SRL	D	NEXT BIT
306B	10E7	00740		DJNZ	NXTCOL	
306D	C1	00750		POP	BC	RECOVER LINE COUNTER
306E	3E00	00760		LD	A,0	
3070	D30B	00770		OUT	(11),A	READ SCREEN RAM
3072	C9	00780		RET		
		00790				
3073	3E1B	00800	PRNTGR	LD	A,27	ESC
3075	CD4580	00810		CALL	8045H	
3078	3E4B	00820		LD	A,75	K
307A	CD4580	00830		CALL	8045H	
307D	3E00	00840		LD	A,0	CHR(0)
307F	CD4580	00850		CALL	8045H	
3082	3E02	00860		LD	A,2	CHR(2)
3084	CD4580	00870		CALL	8045H	
3087	C9	00880		RET		
		00890				
3088	3E02	00900	OUTPP	LD	A,2H	
308A	32E300	00910		LD	(0E3H),A	
308D	3E00	00920		LD	A,0	
308F	320901	00930		LD	(109H),A	
3092	3E0F	00940		LD	A,0FH	
3094	D301	00950		OUT	(1),A	
3096	3E80	00960		LD	A,80H	
3098	D301	00970		OUT	(1),A	
309A	C9	00980		RET		
		00990				
309B	3E1B	01000	LSPACE	LD	A,27	ESC
309D	CD4580	01010		CALL	8045H	
30A0	3E31	01020		LD	A,49	1
30A2	CD4580	01030		CALL	8045H	
30A5	C9	01040		RET		
		01050				
		01060	END			
0000		01060				
00000	Total errors					
BITOFF	3060	NXTROW	3058	NXTCOL	3054	FINISH
BOT	3040	PRINT	3048	TOP	3026	NXTCHR
PRNTGR	3073	LINE	3010	LSPACE	309B	OUTPP
						3088

This program cannot be assembled at F400H because this part of RAM is switched when the character ROM is accessed and the CPU would then try to execute character data!

All registers used by the program are first saved, and OUTPP called if required. HL is loaded with the first screen position, C is set to count the lines as they are printed, and B to count the characters on each line. LSPACE sets the line spacing and PRNTGR sets up graphics mode for 512 bytes as described above. Each character is read from its screen position, multiplied by sixteen and added to either 61440 (F000H) or 61448 depending on whether the tops or bottoms of characters are being printed. The former number gives the address of the first row byte and the latter, the ninth. Which is selected is determined by whether the line counter C is odd or even.

The subroutine PRINT prints one half of the character, either top or bottom. The first two instructions in this subroutine set the ROM read switch as described above. The D register is loaded with 128, or 10000000 and the C register is SLA'd which places a 0 in the Bit 0 position. HL contains the address of the first or ninth row of the character and the byte from that address is ANDed with D. This masks out all but Bit 7, setting the Z flag if the bit is 0 and resetting it if it is 1. In the latter case Bit 0 of the C register is changed from 0 to 1. HL is incremented to read the next row of the character and C is shifted left again, preparing it to receive the next bit. This procedure is performed eight times until a byte has been constructed in the C register from the eight Bit 7's of the row bytes. See Fig 8-2.

Fig. 8-2



The byte in C is then sent to the printer. This whole procedure is then repeated except that the D register is shifted right, resulting in 01000000. Consequently Bit 6 of each row byte is now read and constructed into a new byte in C. This byte is then sent to the printer which prints it to the right of the first byte. This process is repeated for a total of eight times, by which time the entire top or bottom of the character has been printed. Note that the B register is used as a counter for both the above procedures and BC must therefore be PUSHed and POP'd before and after each procedure. HL is also PUSHed and POP'd to preserve the character address while it is being incremented. At the end of this subroutine the ROM read switch is returned to normal.

After the top or bottom of each character has been printed, HL is POP'd again to recover the screen position and incremented so that the next half character can be printed. After this has occurred 64 times a CR and LF are sent to the printer to start on the next line. At this point Bit 0 of the line counter C is tested again to determine whether tops or bottoms of the characters have been printed. If tops, 64 is subtracted from HL so that the bottoms of the characters are

printed. C is decremented at the end of each line and the program finishes when C is zero.

Screen print-outs can be obtained with this program in the same way as with the previous program, but you will need to assemble it into a safe part of RAM, say 3000H for 16K Microbees or 7000H for 32K models, to avoid destruction by the graphics program. If assembled at 3000H, to obtain a print-out use P = USR (12288) from a BASIC program or CALL 3000H from a machine language program.

writes over the interrupt service address used by the parallel port, preventing the screen dump from operating with parallel printers. To overcome this problem, POKE 128,8 after the HIRES command.

# Everything a micro should be for software

## Diskette Software

Borland Turbo Pascal	\$69.50
Adventure Pak 1	\$29.95
Bee Casino	\$29.95
Bee Scientist 1	\$39.95
Bee Scientist 2	\$29.95
Learning Can Be Fun	\$39.95
Disk Catalog System	\$19.95
Geo-tech Drawing	\$21.95
Sydney Approach	\$19.95
Temple of Azragor	\$19.95
Castle of Eldritch	\$19.95
Market Simulation	\$19.95
Sketchpad	\$19.95
Capture	\$19.95
Chess/Chess Tutor	\$19.95
Chopper Pilot	\$19.95
Defender	\$19.95
Dungeons of Khan	\$19.95
Electronic Hobby Mate	\$19.95
Eureka	\$19.95
Eye of Min	\$19.95
Genius and Insanity	\$19.95
Geograbee	\$19.95
Histrobee	\$19.95
Miner	\$19.95
Ring of Doom	\$19.95
Robot Man	\$19.95
Sabre of Sultar	\$19.95
Scrambler	\$19.95
Space Invaders	\$19.95
Survivor	\$19.95
The Valley & Caves	\$19.95
Typing Tutor/Hints	\$19.95
Viper	\$19.95
Wonderwords	\$19.95
Word Adventure	\$19.95

## Cassette Software

Adventure in 3D	Eye of Min
Amazon Search	Frontier Adventure
A-Reading-All	Geograbee
Atlantic Sea Battle	Geo-Tech Drawing (1 & 2)
Azragor — Temple of	Geo-Tech Drawing (3 & 4)
Bee Piano	General Ledger
Big Character VDU	Genetic Inheritance
Bill the Barman	Genius & Insanity
Biorythms/Printer Pac	Golf
Black Jack	Graphic Games (Vol 1)
Busy Calc	Graphic Lander
Capture	Graphic Life
Cannibals and Missionaries	Gridfire
Cave Hunt	Hustle
Chase Plus	Insurance Inventory
Chemical Equations	Introduction to Chemistry
Chess/Chess Tutor	Joysub Justle
Chopper Pilot	Keplers Law
Complex	Keyboard Tutorial
Computer Modelling	Lazer Blazer
Concentration	Learning Can Be Fun 1A
Craps and Roulette	Learning Can Be Fun 1B
Coulombs Law	Learning Can Be Fun 1C
Data Base	Learning Can Be Fun 2A
Debug (16 + 32K)	Learning Can Be Fun 2B
Depth Charge	Learning Can Be Fun 2C
Defender	Logo 16 & 32
Destroyer	Marbles and Jail Bait
Draughts	Market Simulation
Dungeons of Khan	Maths Adventure
Electro Mate	Matrix Calculator
Eliza	Metric Converter
El-Reading-OU	Millikans Experiment
Escape from Colditz	Miner
Escape Key (16 + 32K)	Morse Code (Tutorial)
Eureka	Museum Adventure

Moving with Maths
Noughts and Crosses
One Day Cricket
P.C.G. Sampler
Pools and Lotto
Projectiles
Prospector
Quest for the Castle Eldritch
Reels
Ring of Doom
Robotman (Colour)
Sabre of Sultar
Scrambler
Sea Dog
Search and Destroy Wumpus
Sketch Pad
Space Lanes
Speed Reading
Spelling Wars 1 & 2
Stat Tutorial
Stat Pack Ver 1 to 4
Survivor
Space Invaders
Space Lanes
Super Disassembler
Sword Quest
Sydney Approach
Target/Z Trek
Teachers Busy Bee
Text Formatter
The Valley and Caves
Turtle
Typing Tutor/Basic Hints
Vectors & Graphing
Video Maze
Viper
Work A Bee
Word Adventure
Wonderwords
Yahtze

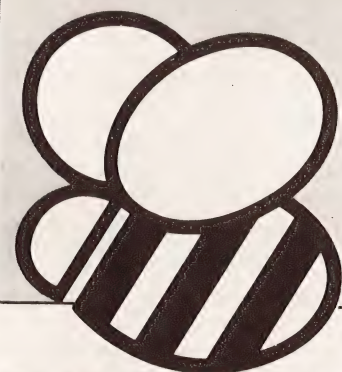
**"Software  
for  
Education  
Games  
Utility  
and  
Business"**

Cassettes .....\$12.50

Diskettes from .....\$19.95

**bankcard**  
welcome here

**DIRECT ORDERS**  
**PHONE (02)487 2711**  
**TELEX AA72767**  
**APPLIED TECHNOLOGY**  
**RETAIL PTY LTD**



## microbee technology centres

N.S.W.  
1 Pattison Ave,  
Waitara 2077, N.S.W.  
Phone (02) 487 2711

VIC.  
729 Glenferrie Rd,  
Hawthorn 3122, Vic.  
Phone (03) 819 5288

W.A.  
141 Stirling Highway,  
Nedlands, W.A.  
Phone (09) 386 8289

S.A.  
151 Unley Road,  
Unley 5061, S.A.  
Phone (08) 272 1384

QLD.  
455 Logan Road,  
Stones Corner, Qld 4120  
Phone (07) 394 3688

FACTORY Koala Crescent, West Gosford 2250.  
Phone (043) 24 2711

## Authorised Dealers in:

•Coffs Harbour  
(066)52 7530

•Lismore  
(066)21 8180

•Melbourne  
(03)233 9665

•Townsville  
(077)75 4000

•Cairns  
(070)51 4341

•Milton  
(07)371 6888

•Port Lincoln  
(086)82 2666

•Mount Tom Price  
(091)89 1383

•Launceston  
(003)31 7890

•Wauchope  
(065)85 3311

**microbee**  
Designed and manufactured in Australia

# MORE ARITHMETIC

*Negative numbers, multiple precision, and decimal numbers.*

All the arithmetic we have so far performed has been with *positive integers*. We have not discussed negative numbers or non-integer numbers (real numbers or numbers with a decimal point). We have performed subtraction of one positive integer from another and tested for a negative result, but have had no way of determining what the result was. We will now look at these subjects in detail.

## Subtraction

As you are now aware the CPU has severe limitations on the kind of tasks it can do, although what it can do it does very fast and very reliably. Although addition of two binary numbers can be achieved with a simple logical process, subtraction is much more difficult. For this reason subtractions are performed by the CPU by using an *addition* process. Let us see how this works. Consider the following addition:

$$\begin{array}{r} 01010011 \quad 83 \\ + 10000000 \quad 256 \\ \hline 101010011 \end{array}$$

We have added 256, which is a 9-bit binary number, to 83, which is an 8-bit binary number. The result is 339. However, if we *ignore the 9th bit* the result is 01010011, or 83. Of course, an eight-bit register cannot contain 256, so the above sum is not possible using eight-bit registers. However, suppose that instead of adding 256 we added 255:

$$\begin{array}{r} 01010011 \quad 83 \\ + 11111111 \quad 255 \\ \hline 101010010 \quad 82 + \text{Carry} \end{array}$$

The answer is 338, but if we ignore the 9th bit the result is 01010010, or 82. Therefore, we have effectively subtracted 1. If we do this sum by placing 83 in the A register and adding 255 to it, the number in the A register after the addition will be 82 and the C flag will be set. This, in fact, is how the CPU would subtract 1 from 83. To subtract 2, 254 would be added, and so on.

Putting this more generally, to subtract y from x the CPU adds (256-y) to x and ignores the Carry generated.

On the face of it, it seems that we have replaced one difficult task, that of subtracting y from x, with another equally difficult task, that of subtracting y from 256. Well, it turns out that this latter task is quite easy for the CPU. The number 256-y is known as the 'two's complement' of y. To calculate it, the CPU

- (1) complements all bits of the number y
- and (2) adds 1.

For example, the two's complement of 1 is obtained as follows:

```

00000001 1
11111110 complement all bits
+ 00000001 add 1
= 11111111 255 (= two's complement of 1)

```

This number, as we already know, is 255, or 256-1.

For those mathematically inclined, there is a formal proof of this process in Appendix 7.

Let us now look at an example of the complete subtraction process. Suppose we

```

LD    A,150
SUB   100

```

After execution of the first instruction the A register will contain

```

A 10010110 150

```

When the SUB 100 instruction is encountered, the two's complement of 100 is first calculated as follows:

```

01100100 100
10011011 complement all bits
+ 00000001 add 1
= 10011100 two's complement of 100
              (= 256-100 = 156)

```

This number is then added to the A register:

```

A 10010110 150
+ 10011100 two's complement of 100
= 100110010 50

```

The A register cannot contain the 9th bit which has been generated and the result, as we required, is 50.

### Testing for a Negative Answer

The process which we have described above to perform subtractions can be put algebraically

$$x + (256-y) = x - y + 256$$

If  $x$  is greater than  $y$ ,  $x-y$  is positive and the +256 remaining is represented by the setting of the C flag as we have already seen. However, if  $x$  is less than  $y$ ,  $x-y$  is negative and the result is less than 256. *Consequently the C flag will not be set.* We now have a way of detecting whether the result of the subtraction  $x-y$  is positive or negative. It would appear that,

if  $x-y$  is positive the C flag is set  
 if  $x-y$  is negative the C flag is reset.

However, this is not the case, because after a subtraction has been performed *the CPU automatically complements the C flag.* Consequently, after a SUB or SBC operation,

if the result is positive (or zero) C is reset  
 if the result is negative C is set.

We introduced this principle in Chapter 2 when we were discussing the flag register and we have used it in many of the programs through the book.

### Representation of Negative Numbers

We have discovered how the CPU subtracts one positive number from another and how to detect if the answer is positive or negative. However, the answer is only useful to us if it is positive because, up to now, we have had no way of representing negative numbers. We will now look at this problem.

We have established that to subtract one positive number from another we add the two's complement of the first number to the second and ignore the C flag. That is,

$$x - y \text{ is equivalent to } x + (256-y)$$

But subtracting a positive number is the same as adding a negative number. That is,

$$x - y \text{ is equivalent to } x + (-y)$$

By comparing the two expressions above it is apparent that the two's complement of a number (i.e., 256- $y$ ) can be used as a negative representation of that number. That is,

$$-y = (256-y)$$

so long as we ignore the C flag. For example,

-1 can be represented by 255 (11111111)  
 -2 can be represented by 254 (11111110)  
 etc.

That is all very well, but how do we decide if, for instance, 11111111 represents 255 or -1? Well, the decision is up to the programmer. If you know that your arithmetic will not involve negative numbers, you can simply regard all numbers as positive. You may also test for a negative result after a subtraction, as we have seen, so long as you do not wish to know what the result is.

If, however, you want to use negative as well as positive numbers in your arithmetic you must allocate some of the 255 numbers available in a single register to represent negative numbers. In fact it is convenient to allocate 128 to 255 for this purpose, so that,

-1 is represented by 255 (11111111)  
 -2 is represented by 254 (11111110)  
 —  
 —  
 —  
 -128 is represented by 128 (10000000)

These numbers are chosen because they all have bit 7 set. We can therefore test whether a number is positive or negative by testing this bit. The remaining numbers, 0 to 127, are positive numbers, as usual.

If we place our new system of numbers on a 'totem pole' it will look like this:

	127	01111111
	3	00000011
	2	00000010
	1	00000001
C flag set →	0	00000000
	-1	11111111
	-2	11111110
	-3	11111101
	-127	10000001
	-128	10000000

Addition and subtraction of positive and negative numbers can be performed in exactly the same way as with positive numbers only. This is because we have chosen to ignore the Carry that is generated as we move up the totem pole from -1 to 0. Let us look at some examples.

Addition with a positive answer:

$$\begin{array}{r} -1 \quad 11111111 \\ +2 \quad +00000010 \\ \hline =1 \quad 00000001 \end{array}$$

Addition with a negative answer:

$$\begin{array}{r} -1 \quad 11111111 \\ +(-2) \quad +11111110 \\ \hline =-3 \quad 11111101 \end{array}$$

Subtraction with a positive answer:

$$\begin{array}{r} 2 \quad 00000010 \\ -1 \quad +11111111 \\ \hline =1 \quad 00000001 \end{array}$$

as before.

Subtraction with a negative answer:

$$\begin{array}{r} 1 \quad 00000001 \\ -2 \quad +11111110 \\ \hline =-1 \quad 11111111 \end{array}$$

Subtracting a negative number:

The two's complement of a negative number is its positive equivalent. For example, the two's complement of -2 is 2, as shown below:

$$\begin{array}{r} 11111110 \quad -2 \\ 00000001 \quad \text{complement all bits} \\ +00000001 \quad \text{add 1} \\ \hline =00000010 \quad \text{two's complement of -2 (= 2)} \end{array}$$

This is because  $256 - (256 - 2) = 2$ .

Therefore, subtracting -2 from 1 becomes:

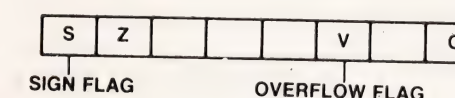
$$\begin{array}{r} 1 \quad 00000001 \\ -(-2) \quad +00000010 \\ \hline =3 \quad 00000011 \end{array}$$

Which is really the addition of 1 and 2. This, of course, gives the correct answer.

Note that with two's complement representation of negative numbers we have the same kind of limitation that exists when using all positive numbers, *that both numbers and the result must be in the range of the system, that is, between -128 and +127.*

### Flags

There are two flags in the Flag register devoted to two's complement arithmetic.



The S, or sign flag indicates whether the result of an arithmetic operation is a positive or negative number. It is simply a copy of bit 7 of the A register which, as we saw, is 1 if the number is negative and 0 if it is positive. The conditions are represented in the source code by P for Plus and M for Minus. For example,

JP P,1234H ;IF S = 0  
or JP M,1234H ;IF S = 1

The V, or Overflow flag is set whenever the result of an arithmetic operation is outside the range -128 to +127. This condition is represented in the source code by PE which stands for Parity Even. (This is because the same flag is used for a parity check on non-arithmetic operations which we will not discuss). The instruction

JP PE,error

will detect the overflow. The V flag reset condition is represented by PO which stands for Parity Odd.

Note that when using two's complement notation the C flag has no meaning and is therefore completely ignored.

### Register Pairs

Exactly the same principles as above apply to register pairs (and the 16-bit Index registers) when using two's complement representation of negative numbers. The range of numbers available is then between -32768 and +32767. After an ADC or SBC (not ADD) operation the V flag (PE) is set if the result is outside this range, and the S flag reflects bit 15 of the HL register pair.

### The Assembler

The Assembler uses two's complement representation of negative numbers. That is,

LD B,-1

will assemble as 06FF, where 06 is the code for LD B and FF is -1 as a one-byte number. Similarly,

```
LD DE,-1
```

will assemble as 11FFFF, where 11 is the code for LD DE and FFFF is -1 as a two-byte number.

Likewise, the JR instruction represents the number of addresses to be jumped forward by positive numbers and the number backwards by negative, or two's complement numbers. This sets the limit to the number of addresses which can be jumped to +127 or -128 from the end of the JR instruction, which, because the instruction is two bytes long is equivalent to 129 forward and 126 backward, counting from the start of the instruction.

### Multiple Precision Arithmetic

All the arithmetic we have performed so far has been on one or two-byte numbers which, as we have seen, limits the size of the numbers to 65535 for positive numbers only, or -32768 to +32767 for a mixture of positive and negative numbers. We will now see how we can extend the range of numbers by using more than two bytes to define each number, which is referred to as 'multiple precision'. (Strictly speaking, multiple precision means using more than one byte per number, but the Z80 handles two-byte numbers automatically without the need for the technique described below).

A three-byte number will contain 24 bits and will have a maximum value of  $2^{24}-1$  or 16777215 for positive numbers only, or from -8388608 to +8388607 using two's complement notation. We can use as many bytes as necessary to contain the size of number we require. However, because the CPU registers can only contain two-byte numbers, multiple precision numbers must be stored, byte by byte, in RAM. Any arithmetic must be performed on them one byte at a time by loading each byte into the CPU, performing the required operation, and loading the result back into RAM.

Let us take as an example the addition of two three-byte numbers, as follows:

EACH BYTE	HIGH BYTE		MIDDLE BYTE		LOW BYTE	
	Bit 7	Bit 0	Bit 7	Bit 0	Bit 7	Bit 0
COMPLETE NUMBER	Bit 23	Bit 16	Bit 15	Bit 8	Bit 7	Bit 0
	1 0 1 1 1 0 1 0		1 0 1 0 1 0 0 0		1 0 0 1 1 0 0 0	
+	0 1 0 0 0 0 0 1		1 1 1 0 1 0 1 0		1 0 1 1 0 1 1 0	
=	1 1 1 1 1 1 0 0		1 0 0 1 0 0 1 1		0 1 0 0 1 1 1 0	

If we did this sum by hand we would first add bits 0, then bits 1 etc as we saw in Chapter 1, adding any carry generated into the next highest bit position. The CPU will do this automatically for us up to bits 7 if we simply add the two low-order bytes with an ADD instruction. However, when we add bits 8, which will be bits 0 of the next byte, we must include any carry generated by the addition of bits 7. We know that the C flag is set or reset according to this result, so we must add in the contents of the C flag with the addition of the middle bytes. This is accomplished automatically by the ADC (Add with Carry) instruction. Similarly, the addition of the high bytes must include the C flag which has been conditioned (set or reset) by the addition of the middle bytes.

Below is a routine which adds two three-byte numbers in the manner just described. It is assumed that the numbers are stored in successive RAM addresses

in low byte, middle byte, high byte order in the conventional Z80 manner and that the low byte addresses of the two numbers are held in the IX and IY registers respectively. The routine adds the second number to the first number, replacing the first number in RAM with the result of the addition, just as the instruction ADD HL,DE adds DE to HL and replaces HL with the result.

```
LD A,(IX) ;LOW BYTE FIRST NO.
ADD A,(IY) ;ADD LOW BYTE SECOND NO.
LD (IX),A ;RESULT INTO RAM
LD A,(IX+1) ;MIDDLE BYTE 1ST NO.
ADC A,(IY+1) ;ADD MIDDLE BYTE 2ND NO. WITH CARRY
LD (IX+1),A ;RESULT INTO RAM
LD A,(IX+2) ;ETC
ADC A,(IY+2)
LD (IX+2),A
```

Note that the first addition, that of the low bytes, is done with an ADD instruction. We do not want to add a carry left in the C flag from some previous operation. The next addition uses the ADC instruction which adds in the contents of the C flag conditioned by the ADD instruction as described above. Note that the two intervening LD instructions do not affect the C flag. Note also that we have used the indexed addressing capabilities of the IX and IY registers to produce an efficient source code.

The same basic routine can be used to perform a three-byte subtraction. All that is necessary is to change the ADD A,(IY) instruction to SUB (IY) (the A is understood in the SUB instruction) and the ADC instructions to SBC. The SBC instruction adds the contents of the C flag to the number to be subtracted before the subtraction occurs. This corresponds to what we would do by hand; if the subtraction of the low bytes produced a negative result and consequently set the C flag, we would subtract one more from the next subtraction to compensate for the 'borrow'.

### Representation of Real Numbers

Now we come to the question of how to represent real numbers, or numbers with a decimal point, in the computer. The Z80 has no built-in method of handling real numbers so we must find a way of representing such numbers and performing arithmetic on them within the limitations of the 8- or 16-bit integer arithmetic which the Z80 is able to do.

There is no single method of doing this and the choice of method is up to the programmer. However we will briefly describe one method without going into detail of how the arithmetic is performed.

The problem is to represent such numbers as 123.4 or 0.0567. It would be possible to allocate a number of bytes for digits to the right of the decimal point and a number for digits to the left, but this would waste memory space and unnecessarily limit the range of numbers we could use. A better way is to represent each number by a *mantissa* and an *exponent*, like this:

$123.4 = 1234 \times 10^{-1}$  in which 1234 is the mantissa and -1 is the exponent, and

$0.0567 = 567 \times 10^{-4}$  in which 567 is the mantissa and -4 is the exponent.

A still better way is to use a *normalised* mantissa, like this,

and  $123.4 = .1234 \times 10^3$

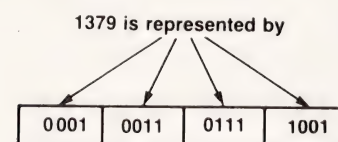
and  $0.0567 = .567 \times 10^{-1}$

in which the mantissa is formed by moving the decimal point to the left of the significant digits. This method allows us to allocate a certain number of bytes to the mantissa depending on how many significant digits we want to be able to handle. If we decide to use three bytes for this purpose we will be able to handle mantissas between -8388608 and +8388607, using two's complement notation. We also need some way to represent the exponent and if we allocate one byte for this purpose we can handle exponents from -128 to +127. In a binary number the exponent is actually a power of 2, so the whole system can represent positive numbers from  $8388607 \times 2^{-128}$  to  $8388607 \times 2^{127}$  and negative numbers from  $-8388608 \times 2^{-128}$  to  $-8388608 \times 2^{127}$ , which is quite a range of numbers! Here is the layout for this system:

BYTE 1	BYTE 2	BYTE 3	BYTE 4
EXPONENT		MANTISSA	
-128		-8388608	
TO		TO	
+127		+8388607	

### BCD Representation

One other way of representing numbers remains to be discussed, that is, Binary Coded Decimal or BCD. In this method each individual decimal digit is represented by its binary equivalent and occupies half a byte, or a nibble. For example,



which is 1379 hex. Because each decimal digit occupies half a byte, the number of digits which can be held is twice the number of bytes reserved for the task. For example, an accuracy of eight significant digits would require four bytes for each number.

Note that in BCD numbers the nibbles 1010 to 1111, corresponding to hex A,B,C,D,E, and F, do not occur because there are no equivalent decimal digits. This gives rise to a problem with BCD arithmetic, as we shall now see.

### BCD Arithmetic

Let us look at the addition of two BCD numbers, each of two digits, and thus each one byte long.

$$\begin{array}{r}
 0001|0010 \text{ represents } 12 \\
 + 0100|0101 \text{ represents } 45 \\
 \hline
 = 0101|0111 \text{ represents } 57
 \end{array}$$

Binary addition results in a number which represents the correct answer in BCD form.

Now let us look at another example.

$$\begin{array}{r}
 0010|0101 \quad 25 \\
 + 0110|0111 \quad + 67 \\
 \hline
 = 1000|1100 \quad = 8?
 \end{array}$$

When these two bytes are added the low (right hand) nibble of the result contains 1100 which has no meaning in BCD representation, as we noted above. Moreover, the high nibble contains 1000 (= 8) which is wrong. The reason is that the addition resulted in a number greater than 9. To overcome this problem, any nibble which contains more than 9 (1001) after an addition must be corrected by adding 6 (0110) to 'skip over' the six illegal BCD numbers. Let us try our addition again using this technique:

$$\begin{array}{r}
 0010|0101 \quad 25 \\
 + 0110|0111 \quad + 67 \\
 \hline
 = 1000|1100 \quad > 9 \\
 + \quad \quad 0110 \quad \text{so add 6} \\
 \hline
 = 1001|0010 = 92 \\
 \text{carry}
 \end{array}$$

This gives the correct answer. Note that the high nibble is now correct due to the carry generated by the addition of the 6. Now consider the addition:

$$\begin{array}{r}
 0010|1000 \quad 28 \\
 + 0110|1001 \quad + 69 \\
 \hline
 = 1001|0001 \quad = 91 \text{ not } > 9
 \end{array}$$

We've got the wrong answer again! If you analyse it you will see that, even though the low nibble addition (8 + 9) produced a number greater than 9, the resultant nibble was only 1 hex because a 'half carry' was generated from this nibble to the high nibble (ie from bit 3 to bit 4). However, we can use this fact to signal that the addition of 6 is required. Let us try again:

$$\begin{array}{r}
 0010|1000 \quad 28 \\
 + 0110|1001 \quad + 69 \\
 \hline
 = 1001|0001 \quad \text{half carry} \\
 + \quad \quad 0110 \quad \text{so add 6} \\
 \hline
 = 1001|0111 \quad = 97
 \end{array}$$

We now have the correct answer. To summarise, when the addition of two nibbles results in a number between 9 and 15, or a half carry, 6 must be added to the result to correct it.

All this looks complicated but, in practice, the complication has been overcome for us by the Z80 DAA instruction. This Decimal Adjust Accumulator instruction adjusts each nibble of the A register when necessary to give the correct answer in BCD form after add or subtract instructions.

Here is a program to demonstrate the operation of the DAA instruction. It adds 1234 and 5678 and displays the result, 6912 on the screen.

```

ORG 400H
LD A,34H
LD B,78H
ADD A,B      ;ADD LOW BYTES
DAA          ;ADJUST FOR BCD
LD L,A       ;STORE SUM IN L
LD A,12H

```

```

LD    B,56H
ADC   A,B      ;ADD HIGH BYTES
DAA                   ;ADJUST FOR BCD
LD    H,A      ;STORE SUM IN H
LD    DE,0F23AH; SCREEN POSITION
CALL  0DCB8H   ;DISPLAY HL AT DE
RET
END

```

The numbers are to be in BCD form, so the number 1234 is represented by two bytes as follows:

12H	34H
-----	-----

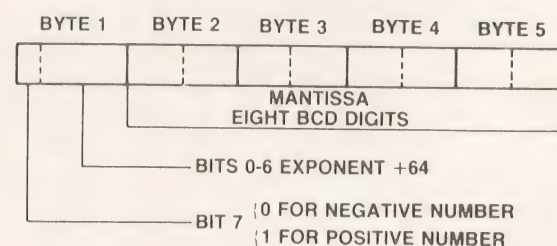
These bytes would normally be stored in RAM, but the program loads the bytes directly into the A and B registers for simplicity. The low byte of the first number is loaded into the A register and that of the second number into the B register, and the two are added in the A register. The DAA instruction adjusts the result as we have seen and the adjusted result is then loaded into L. The high bytes are treated similarly except that an ADC instruction is used. The reason is exactly the same as we saw with multiple precision arithmetic; that is, to add any carry generated by the low byte addition. The adjusted result of the high byte addition is then loaded into H so that HL now contains the two bytes of the sum in BCD form.

So that we can see if we have the correct answer, we load DE with a suitable screen position and CALL 0DCB8H. This is a useful ROM routine within the EDASM which displays the contents of HL in hex form at screen position DE. If you run the program you will find the answer 6912 at the right centre of the screen.

### Real Numbers in BCD Form

We saw previously how real numbers can be represented by an extension of two's complement integer form by specifying a mantissa and an exponent. We can also do the same thing with numbers in BCD form. As before, we shall use as many bytes for the mantissa as are necessary for the number of significant digits we require, and one extra byte for the exponent. If, for instance, we required eight significant digits we would need four bytes for the mantissa, as we have already seen, and one for the exponent, making a total of five.

In our previous system we used two's complement notation for both the mantissa and the exponent, allowing us to specify positive and negative numbers for both. In BCD form negative numbers must be specified with a separate sign bit and we will use one bit of the exponent byte for this purpose. The other seven bits of this byte will specify the exponent as a seven-bit number which has a normal range of 0 to 127. To allow for negative exponents, 64 will be subtracted from the number, giving an exponent range of -64 to +63. Here is a layout of the scheme.



As an example, the number -.012345678, which is  $-.12345678 \times 10^{-1}$  would be represented as

00111111	12H	34H	56H	78H
-	-1			

This, in fact, is how the Microbee represents real numbers.

# Everything a micro should be

Applied Technology  
Retail Pty Ltd

Sydney (02) 487 2711  
Melbourne (03) 819 5288  
Brisbane (07) 394 3688  
Adelaide (08) 272 1384  
Perth (09) 386 8289  
Gosford (043) 24 4711



**microbee®**

Top Selling Australian Computer

DEC	HEX	BINARY	DEC	HEX	BINARY	DECIMAL	HEX	BINARY	DECIMAL	HEX	BINARY		
0	00	00000000	64	40	01000000	-128	128	80	10000000	-64	192	C0	11000000
1	01	00000001	65	41	01000001	-127	129	81	10000001	-63	193	C1	11000001
2	02	00000010	66	42	01000010	-126	130	82	10000010	-62	194	C2	11000010
3	03	00000011	67	43	01000011	-125	131	83	10000011	-61	195	C3	11000011
4	04	00000100	68	44	01000100	-124	132	84	10000100	-60	196	C4	11000100
5	05	00000101	69	45	01000101	-123	133	85	10000101	-59	197	C5	11000101
6	06	00000110	70	46	01000110	-122	134	86	10000110	-58	198	C6	11000110
7	07	00000111	71	47	01000111	-121	135	87	10000111	-57	199	C7	11000111
8	08	00001000	72	48	01001000	-120	136	88	10001000	-56	200	C8	11001000
9	09	00001001	73	49	01001001	-119	137	89	10001001	-55	201	C9	11001001
10	0A	00001010	74	4A	01001010	-118	138	8A	10001010	-54	202	CA	11001010
11	0B	00001011	75	4B	01001011	-117	139	8B	10001011	-53	203	CB	11001011
12	0C	00001100	76	4C	01001100	-116	140	8C	10001100	-52	204	CC	11001100
13	0D	00001101	77	4D	01001101	-115	141	8D	10001101	-51	205	CD	11001101
14	0E	00001110	78	4E	01001110	-114	142	8E	10001110	-50	206	CE	11001110
15	0F	00001111	79	4F	01001111	-113	143	8F	10001111	-49	207	CF	11001111
16	10	00010000	80	50	01010000	-112	144	90	10010000	-48	208	D0	11010000
17	11	00010001	81	51	01010001	-111	145	91	10010001	-47	209	D1	11010001
18	12	00010010	82	52	01010010	-110	146	92	10010010	-46	210	D2	11010010
19	13	00010011	83	53	01010011	-109	147	93	10010011	-45	211	D3	11010011
20	14	00010100	84	54	01010100	-108	148	94	10010100	-44	212	D4	11010100
21	15	00010101	85	55	01010101	-107	149	95	10010101	-43	213	D5	11010101
22	16	00010110	86	56	01010110	-106	150	96	10010110	-42	214	D6	11010110
23	17	00010111	87	57	01010111	-105	151	97	10010111	-41	215	D7	11010111
24	18	00011000	88	58	01011000	-104	152	98	10011000	-40	216	D8	11011000
25	19	00011001	89	59	01011001	-103	153	99	10011001	-39	217	D9	11011001
26	1A	00011010	90	5A	01011010	-102	154	9A	10011010	-38	218	DA	11011010
27	1B	00011011	91	5B	01011011	-101	155	9B	10011011	-37	219	DB	11011011
28	1C	00011100	92	5C	01011100	-100	156	9C	10011100	-36	220	DC	11011100
29	1D	00011101	93	5D	01011101	-99	157	9D	10011101	-35	221	DD	11011101
30	1E	00011110	94	5E	01011110	-98	158	9E	10011110	-34	222	DE	11011110
31	1F	00011111	95	5F	01011111	-97	159	9F	10011111	-33	223	DF	11011111
32	20	00100000	96	60	01100000	-96	160	A0	10100000	-32	224	E0	11100000
33	21	00100001	97	61	01100001	-95	161	A1	10100001	-31	225	E1	11100001
34	22	00100010	98	62	01100010	-94	162	A2	10100010	-30	226	E2	11100010
35	23	00100011	99	63	01100011	-93	163	A3	10100011	-29	227	E3	11100011
36	24	00100100	100	64	01100100	-92	164	A4	10100100	-28	228	E4	11100100
37	25	00100101	101	65	01100101	-91	165	A5	10100101	-27	229	E5	11100101
38	26	00100110	102	66	01100110	-90	166	A6	10100110	-26	230	E6	11100110
39	27	00100111	103	67	01100111	-89	167	A7	10100111	-25	231	E7	11100111
40	28	00101000	104	68	01101000	-88	168	A8	10101000	-24	232	E8	11101000
41	29	00101001	105	69	01101001	-87	169	A9	10101001	-23	233	E9	11101001
42	2A	00101010	106	6A	01101010	-86	170	AA	10101010	-22	234	EA	11101010
43	2B	00101011	107	6B	01101011	-85	171	AB	10101011	-21	235	EB	11101011
44	2C	00101100	108	6C	01101100	-84	172	AC	10101100	-20	236	EC	11101100
45	2D	00101101	109	6D	01101101	-83	173	AD	10101101	-19	237	ED	11101101
46	2E	00101110	110	6E	01101110	-82	174	AE	10101110	-18	238	EE	11101110
47	2F	00101111	111	6F	01101111	-81	175	AF	10101111	-17	239	EF	11101111
48	30	00110000	112	70	01110000	-80	176	B0	10110000	-16	240	F0	11110000
49	31	00110001	113	71	01110001	-79	177	B1	10110001	-15	241	F1	11110001
50	32	00110010	114	72	01110010	-78	178	B2	10110010	-14	242	F2	11110010
51	33	00110011	115	73	01110011	-77	179	B3	10110011	-13	243	F3	11110011
52	34	00110100	116	74	01110100	-76	180	B4	10110100	-12	244	F4	11110100
53	35	00110101	117	75	01110101	-75	181	B5	10110101	-11	245	F5	11110101
54	36	00110110	118	76	01110110	-74	182	B6	10110110	-10	246	F6	11110110
55	37	00110111	119	77	01110111	-73	183	B7	10110111	-9	247	F7	11110111
56	38	00111000	120	78	01111000	-72	184	B8	10111000	-8	248	F8	11111000
57	39	00111001	121	79	01111001	-71	185	B9	10111001	-7	249	F9	11111001
58	3A	00111010	122	7A	01111010	-70	186	BA	10111010	-6	250	FA	11111010
59	3B	00111011	123	7B	01111011	-69	187	BB	10111011	-5	251	FB	11111011
60	3C	00111100	124	7C	01111100	-68	188	BC	10111100	-4	252	FC	11111100
61	3D	00111101	125	7D	01111101	-67	189	BD	10111101	-3	253	FD	11111101
62	3E	00111110	126	7E	01111110	-66	190	BE	10111110	-2	254	FE	11111110
63	3F	00111111	127	7F	01111111	-65	191	BF	10111111	-1	255	FF	11111111

## ASCII - HEXADECIMAL - DECIMAL TABLE

ASCII	HEX	DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX	DECIMAL
NUL	00	0	+	2B	43	V	56	86
SOH	01	1	,	2C	44	W	57	87
STX	02	2	-	2D	45	X	58	88
ETX	03	3	.	2E	46	Y	59	89
EOT	04	4	/	2F	47	Z	5A	90
END	05	5	0	30	48	[	5B	91
ACK	06	6	1	31	49	\	5C	92
BEL	07	7	2	32	50	]	5D	93
BS	08	8	3	33	51	^	5E	94
HT	09	9	4	34	52	-	5F	95
LF	0A	10	5	35	53	,	60	96
VT	0B	11	6	36	54	a	61	97
FF	0C	12	7	37	55	b	62	98
CR	0D	13	8	38	56	c	63	99
SO	0E	14	9	39	57	d	64	100
SI	0F	15	:	3A	58	e	65	101
DLE	10	16	;	3B	59	f	66	102
DC1	11	17	<	3C	60	g	67	103
DC2	12	18	=	3D	61	h	68	104
DC3	13	19	>	3E	62	i	69	105
DC4	14	20	?	3F	63	j	6A	106
NAK	15	21	@	40	64	k	6B	107
SYN	16	22	A	41	65	l	6C	108
ETB	17	23	B	42	66	m	6D	109
CAN	18	24	C	43	67	n	6E	110
EM	19	25	D	44	68	o	6F	111
SUB	1A	26	E	45	69	p	70	112
ESC	1B	27	F	46	70	q	71	113
FS	1C	28	G	47	71	r	72	114
GS	1D	29	H	48	72	s	73	115
RS	1E	30	I	49	73	t	74	116
US	1F	31	J	4A	74	u	75	117
SP	20	32	K	4B	75	v	76	118
!	21	33	L	4C	76	w	77	119
"	22	34	M	4D	77	x	78	120
#	23	35	N	4E	78	y	79	121
\$	24	36	O	4F	79	z	7A	122
%	25	37	P	50	80	{	7B	123
&	26	38	Q	51	81		7C	124
'	27	39	R	52	82	}	7D	125
(	28	40	S	53	83	~	7E	126
)	29	41	T	54	84	DEL	7F	127
*	2A	42	U	55	85			

## THE Z80 INSTRUCTION SET

NOTE: One-byte numbers in source code are shown as 12(hex), and two-byte numbers as 1234(hex).

OBJECT CODE	SOURCE CODE	FLAGS AFFECTED
8E	ADC A, (HL)	S Z V C
DD8E12	ADC A, (IX+12)	S Z V C
FD8E12	ADC A, (IY+12)	S Z V C
8F	ADC A, A	S Z V C
88	ADC A, B	S Z V C
89	ADC A, C	S Z V C
8A	ADC A, D	S Z V C
8B	ADC A, E	S Z V C
8C	ADC A, H	S Z V C
8D	ADC A, L	S Z V C
CE12	ADC A, 12	S Z V C
ED4A	ADC HL, BC	S Z V C
ED5A	ADC HL, DE	S Z V C
ED6A	ADC HL, HL	S Z V C
ED7A	ADC HL, SP	S Z V C
86	ADD A, (HL)	S Z V C
DD8612	ADD A, (IX+12)	S Z V C
FD8612	ADD A, (IY+12)	S Z V C
87	ADD A, A	S Z V C
88	ADD A, B	S Z V C
81	ADD A, C	S Z V C
82	ADD A, D	S Z V C
83	ADD A, E	S Z V C
84	ADD A, H	S Z V C
85	ADD A, L	S Z V C
C612	ADD A, 12	S Z V C
89	ADD HL, BC	C
19	ADD HL, DE	C
29	ADD HL, HL	C
39	ADD HL, SP	C
DD09	ADD IX, BC	C
DD19	ADD IX, DE	C
DD29	ADD IX, IX	C
DD39	ADD IX, SP	C
FD09	ADD IY, BC	C
FD19	ADD IY, DE	C
FD29	ADD IY, IY	C
FD39	ADD IY, SP	C
A6	AND (HL)	S Z V C=0
DDA612	AND (IX+12)	S Z V C=0
FDA612	AND (IY+12)	S Z V C=0
A7	AND A	S Z V C=0
A8	AND B	S Z V C=0
A1	AND C	S Z V C=0
A2	AND D	S Z V C=0
A3	AND E	S Z V C=0
A4	AND H	S Z V C=0
A5	AND L	S Z V C=0
E612	AND 12	S Z V C=0
CB46	BIT 0, (HL)	S Z V
DDCB1246	BIT 0, (IX+12)	S Z V
FDCB1246	BIT 0, (IY+12)	S Z V
CB47	BIT 0, A	S Z V
CB48	BIT 0, B	S Z V
CB41	BIT 0, C	S Z V
CB42	BIT 0, D	S Z V
CB43	BIT 0, E	S Z V
CB44	BIT 0, H	S Z V
CB45	BIT 0, L	S Z V
CB4E	BIT 1, (HL)	S Z V
DDCB124E	BIT 1, (IX+12)	S Z V
FDCB124E	BIT 1, (IY+12)	S Z V
CB4F	BIT 1, A	S Z V
CB48	BIT 1, B	S Z V
CB49	BIT 1, C	S Z V
CB4A	BIT 1, D	S Z V
CB4B	BIT 1, E	S Z V
CB4C	BIT 1, H	S Z V
CB4D	BIT 1, L	S Z V
CB56	BIT 2, (HL)	S Z V
DDCB1256	BIT 2, (IX+12)	S Z V
FDCB1256	BIT 2, (IY+12)	S Z V
CB57	BIT 2, A	S Z V
CB58	BIT 2, B	S Z V
CB51	BIT 2, C	S Z V
CB52	BIT 2, D	S Z V
CB53	BIT 2, E	S Z V
CB54	BIT 2, H	S Z V
CB55	BIT 2, L	S Z V
CB5E	BIT 3, (HL)	S Z V
DDCB125E	BIT 3, (IX+12)	S Z V
FDCB125E	BIT 3, (IY+12)	S Z V
CB5F	BIT 3, A	S Z V
CB58	BIT 3, B	S Z V
CB59	BIT 3, C	S Z V
CB5A	BIT 3, D	S Z V
CB5B	BIT 3, E	S Z V
CB5C	BIT 3, H	S Z V
CB5D	BIT 3, L	S Z V
CB66	BIT 4, (HL)	S Z V
DDCB1266	BIT 4, (IX+12)	S Z V
FDCB1266	BIT 4, (IY+12)	S Z V
CB67	BIT 4, A	S Z V
CB68	BIT 4, B	S Z V
CB61	BIT 4, C	S Z V
CB62	BIT 4, D	S Z V
CB63	BIT 4, E	S Z V
CB64	BIT 4, H	S Z V
CB65	BIT 4, L	S Z V
CB6E	BIT 5, (HL)	S Z V
DDCB126E	BIT 5, (IX+12)	S Z V
FDCB126E	BIT 5, (IY+12)	S Z V
CB6F	BIT 5, A	S Z V
CB68	BIT 5, B	S Z V
CB69	BIT 5, C	S Z V
CB6A	BIT 5, D	S Z V
CB6B	BIT 5, E	S Z V
CB6C	BIT 5, H	S Z V
CB6D	BIT 5, L	S Z V
CB76	BIT 6, (HL)	S Z V
DDCB1276	BIT 6, (IX+12)	S Z V
FDCB1276	BIT 6, (IY+12)	S Z V
CB77	BIT 6, A	S Z V
CB78	BIT 6, B	S Z V
CB71	BIT 6, C	S Z V
CB72	BIT 6, D	S Z V
CB73	BIT 6, E	S Z V
CB74	BIT 6, H	S Z V
CB75	BIT 6, L	S Z V
CB7E	BIT 7, (HL)	S Z V
DDCB127E	BIT 7, (IX+12)	S Z V
FDCB127E	BIT 7, (IY+12)	S Z V
CB7F	BIT 7, A	S Z V
CB78	BIT 7, B	S Z V
CB79	BIT 7, C	S Z V
CB7A	BIT 7, D	S Z V
CB7B	BIT 7, E	S Z V
CB7C	BIT 7, H	S Z V
CB7D	BIT 7, L	S Z V
DC3412	CALL C, 1234	
FC3412	CALL M, 1234	
D43412	CALL NC, 1234	
C43412	CALL NZ, 1234	
F43412	CALL P, 1234	
EC3412	CALL PE, 1234	
E43412	CALL PO, 1234	
CC3412	CALL Z, 1234	
CD3412	CALL 1234	
3F	CCF	
BE	CP (HL)	S Z V C
DDBE12	CP (IX+12)	S Z V C
FDBe12	CP (IY+12)	S Z V C
BF	CP A	S Z V C
B8	CP B	S Z V C
B9	CP C	S Z V C
BA	CP D	S Z V C
BB	CP E	S Z V C
BC	CP H	S Z V C
BD	CP L	S Z V C
FE12	CP 12	S Z V C
EDA9	CPD	S Z V
EDB9	CPDR	S Z V
EDB1	CPIR	S Z V
EDA1	CPI	S Z V
2F	CPL	
27	DAA	S Z V
35	DEC (HL)	S Z V
DD3512	DEC (IX+12)	S Z V
FD3512	DEC (IY+12)	S Z V
3D	DEC A	S Z V
05	DEC B	S Z V
0B	DEC BC	S Z V



OBJECT CODE	SOURCE CODE	FLAGS AFFECTED			
ED6F	RLD	S Z V	FDCB12E6	SET	4, (IY+12)
CB1E	RR (HL)	S Z V C	CBE7	SET	4, A
DDCB121E	RR (IX+12)	S Z V C	CBE8	SET	4, B
FDCB121E	RR (IY+12)	S Z V C	CBE9	SET	4, C
CB1F	RR A	S Z V C	CBEA	SET	4, D
CB18	RR B	S Z V C	CBE1	SET	4, E
CB19	RR C	S Z V C	CBE2	SET	4, H
CB1A	RR D	S Z V C	CBE3	SET	4, L
CB1B	RR E	S Z V C	CBE4	SET	5, (HL)
CB1C	RR H	S Z V C	CBE5	SET	5, (IX+12)
CB1D	RR L	S Z V C	DDCB12EE	SET	5, (IY+12)
IF	RRA	S Z V C	FDCB12EE	SET	5, A
CB0E	RRC (HL)	S Z V C	CBEF	SET	5, B
DDCB120E	RRC (IX+12)	S Z V C	CBE8	SET	5, C
FDCB120E	RRC (IY+12)	S Z V C	CBE9	SET	5, D
CB0F	RRC A	S Z V C	CBEA	SET	5, E
CB08	RRC B	S Z V C	CBE1	SET	5, H
CB09	RRC C	S Z V C	CBE2	SET	5, L
CB0A	RRC D	S Z V C	CBE3	SET	6, (HL)
CB0B	RRC E	S Z V C	DDCB12F6	SET	6, (IX+12)
CB0C	RRC H	S Z V C	FDCB12F6	SET	6, (IY+12)
CB0D	RRC L	S Z V C	CBE4	SET	6, A
0F	RRCA	S Z V C	CBE5	SET	6, B
ED67	RRD	S Z V	CBE6	SET	6, C
C7	RST 00H		CBE7	SET	6, D
CF	RST 08H		CBE8	SET	6, E
D7	RST 10H		CBE9	SET	6, H
DF	RST 18H		CBEA	SET	6, L
E7	RST 20H		CBE1	SET	7, (HL)
EF	RST 28H		CBE2	SET	7, (IX+12)
F7	RST 30H		CBE3	SET	7, (IY+12)
FF	RST 38H		CBE4	SET	7, A
DE12	SBC A, 12	S Z V C	CBE5	SET	7, B
9E	SBC A, (HL)	S Z V C	CBE6	SET	7, C
DD9E12	SBC A, (IX+12)	S Z V C	CBE7	SET	7, D
FD9E12	SBC A, (IY+12)	S Z V C	CBE8	SET	7, E
9F	SBC A, A	S Z V C	CBE9	SET	7, H
98	SBC A, B	S Z V C	CBEA	SET	7, L
99	SBC A, C	S Z V C	CB26	SLA (HL)	
9A	SBC A, D	S Z V C	DDCB1226	SLA (IX+12)	
9B	SBC A, E	S Z V C	FDCB1226	SLA (IY+12)	
9C	SBC A, H	S Z V C	CB27	SLA A	
9D	SBC A, L	S Z V C	CB28	SLA B	
ED42	SBC HL, BC	S Z V C	CB29	SLA C	
ED52	SBC HL, DE	S Z V C	CB2A	SLA D	
ED62	SBC HL, HL	S Z V C	CB2B	SLA E	
ED72	SBC HL, SP	S Z V C	CB2C	SLA H	
37	SCF	S Z V C	CB2D	SLA L	
CBC6	SET 0, (HL)	C=1	CB2E	SRA (HL)	
DDCB12C6	SET 0, (IX+12)		DDCB122E	SRA (IX+12)	
FDCB12C6	SET 0, (IY+12)		FDCB122E	SRA (IY+12)	
CBC7	SET 0, A		CB2F	SRA A	
CBC8	SET 0, B		CB28	SRA B	
CBC1	SET 0, C		CB29	SRA C	
CBC2	SET 0, D		CB2A	SRA D	
CBC3	SET 0, E		CB2B	SRA E	
CBC4	SET 0, H		CB2C	SRA H	
CBC5	SET 0, L		CB2D	SRA L	
CBCE	SET 1, (HL)		CB2E	SRL (HL)	
DDCB12CE	SET 1, (IX+12)		DDCB123E	SRL (IX+12)	
FDCB12CE	SET 1, (IY+12)		FDCB123E	SRL (IY+12)	
CBCF	SET 1, A		CB3F	SRL A	
CBC8	SET 1, B		CB38	SRL B	
CBC9	SET 1, C		CB39	SRL C	
CBCA	SET 1, D		CB3A	SRL D	
CBCB	SET 1, E		CB3B	SRL E	
CBCC	SET 1, H		CB3C	SRL H	
CBCD	SET 1, L		CB3D	SRL L	
CBDE	SET 2, (HL)		96	SUB (HL)	
DDCB12D6	SET 2, (IX+12)		DD9612	SUB (IX+12)	
FDCB12D6	SET 2, (IY+12)		FD9612	SUB (IY+12)	
CBDF	SET 2, A		97	SUB A	
CBDE	SET 2, B		98	SUB B	
CBDE	SET 2, C		99	SUB C	
CBDE	SET 2, D		9A	SUB D	
CBDE	SET 2, E		9B	SUB E	
CBDE	SET 2, H		9C	SUB H	
CBDE	SET 2, L		9D	SUB L	
CBDE	SET 3, (HL)		9E	SUB (HL)	
DDCB12DE	SET 3, (IX+12)		DD9E12	SUB (IX+12)	
FDCB12DE	SET 3, (IY+12)		FD9E12	SUB (IY+12)	
CBDF	SET 3, A		9F	SUB A	
CBDE	SET 3, B		98	SUB B	
CBDE	SET 3, C		99	SUB C	
CBDE	SET 3, D		9A	SUB D	
CBDE	SET 3, E		9B	SUB E	
CBDE	SET 3, H		9C	SUB H	
CBDE	SET 3, L		9D	SUB L	
CBE6	SET 4, (HL)		9E	SUB (HL)	
DDCB12E6	SET 4, (IX+12)		9F	SUB (IY+12)	

## USEFUL ROUTINES IN THE MICROBEE ROM

8006H	Waits for key to be pressed. Loads key ASCII into A.
8024H	Initialise HIRES.
8027H	Initialise LORES.
8030H	SET dot.
8033H	RESET dot.
8036H	INVERT dot.
8039H	Test for dot set: Z not set, NZ set.
803CH	PLOT line.
8045H	Sends one byte from A to printer.
A50AH	Tests for key pressed whose ASCII value is in A: Z if pressed, NZ if not. Does not wait.
DCB8H	Displays contents of HL in hex at screen position DE.

## TOP LINE ADDRESSES OF PCG CHARACTERS

CHR ADDRESS			HEX DEC								
0	F000	61440	140	F8C0	63680	156	F9C0	63936	172	FAC0	64192
1	F010	61456	141	F8D0	63696	157	F9D0	63952	173	FAD0	64208
:			142	F8E0	63712	158	F9E0	63968	174	FAE0	64224
:			143	F8F0	63728	159	F9F0	63984	175	FAF0	64240
128	F800	63488	144	F900	63744	160	FA00	64000	176	FB00	64256
129	F810	63504	145	F910	63760	161	FA10	64016	177	FB10	64272
130	F820	63520	146	F920	63776	162	FA20	64032	178	FB20	64288
131	F830	63536	147	F930	63792	163	FA30	64048	179	FB30	64304
132	F840	63552	148	F940	63808	164	FA40	64064	180	FB40	64320
133	F850	63568	149	F950	63824	165	FA50	64080	181	FB50	64336
134	F860	63584	150	F960	63840	166	FA60	64096	182	FB60	64352
135	F870	63600	151	F970	63856	167	FA70	64112	183	FB70	64368
136	F880	63616	152	F980	63872	168	FA80	64128	184	FB80	64384
137	F890	63632	153	F990	63888	169	FA90	64144	185	FB90	64400
138	F8A0	63648	154	F9A0	63904	170	FAA0	64160	186	FBA0	64416
139	F8B0	63664	155	F9B0	63920	171	FAB0	64176	187	FBB0	64432

188	FBC0	64448	205	FCD0	64720	222	FDE0	64992	239	FEF0	65264
189	FBD0	64464	206	FCE0	64736	223	FDF0	65008	240	FF00	65280
190	FBE0	64480	207	FCF0	64752	224	FE00	65024	241	FF10	65296
191	FBF0	64496	208	FD00	64768	225	FE10	65040	242	FF20	65312
192	FC00	64512	209	FD10	64784	226	FE20	65056	243	FF30	65328
193	FC10	64528	210	FD20	64800	227	FE30	65072	244	FF40	65344
194	FC20	64544	211	FD30	64816	228	FE40	65088	245	FF50	65360
195	FC30	64560	212	FD40	64832	229	FE50	65104	246	FF60	65376
196	FC40	64576	213	FD50	64848	230	FE60	65120	247	FF70	65392
197	FC50	64592	214	FD60	64864	231	FE70	65136	248	FF80	65408
198	FC60	64608	215	FD70	64880	232	FE80	65152	249	FF90	65424
199	FC70	64624	216	FD80	64896	233	FE90	65168	250	FFA0	65440
200	FC80	64640	217	FD90	64912	234	FEA0	65184	251	FFB0	65456
201	FC90	64656	218	FDA0	64928	235	FEB0	65200	252	FFC0	65472
202	FCA0	64672	219	FDB0	64944	236	FEC0	65216	253	FFD0	65488
203	FCB0	64688	220	FDC0	64960	237	FED0	65232	254	FFE0	65504
204	FCC0	64704	221	FDD0	64976	238	FEE0	65248	255	FFF0	65520

## APPENDIX 6

## SCREEN ADDRESSES — HEX

F000	F020	F03F
F040	F060	F07F
F080	F0A0	F0BF
F0C0	F0E0	F0FF
F100	F120	F13F
F140	F160	F17F
F180	F1A0	F1BF
F1C0	F1E0	F1FF
F200	F220	F23F
F240	F260	F27F
F280	F2A0	F2BF
F2C0	F2E0	F2FF
F300	F320	F33F
F340	F360	F37F
F380	F3A0	F3BF
F3C0	F3E0	F3FF

## APPENDIX 7

Suppose that  $x$  is any eight bit binary number.  
Then,

$$x = a * 2^7 + b * 2^6 + c * 2^5 + \dots + g * 2 + h \quad (1)$$

where  $a, b, c, \dots, h$  may each have the values 0 or 1.  
If each bit is complemented the result, known as the one's complement ( $1cx$ ), is

$$1cx = \bar{a} * 2^7 + \bar{b} * 2^6 + \dots + \bar{g} * 2 + \bar{h} \quad (2)$$

where  $\bar{a}$  represents the complement of  $a$ , etc.  
Adding (1) and (2), because  $a + \bar{a} = 1$ , etc.,

$$x + 1cx = 2^7 + 2^6 + 2^5 + \dots + 2 + 1 \\ = 255$$

Adding 1 to each side,

$$x + 2cx = 256 \text{ where } 2cx \text{ is the two's complement of } x (= 1cx + 1)$$

or,

$$2cx = 256 - x.$$

# WANT TO KNOW MORE?



**Online — the first magazine devoted entirely to you — the microbee owner**

'ONLINE' is the microbee owners journal full of information and features about your favourite personal computer. It is available from your local microbee technology centre at a cost of \$2.50 OR you can subscribe to 'ONLINE' which is published monthly for \$25.00 annually. Why not fill out the coupon now and subscribe? That way

## SUBSCRIBE TODAY AND SAVE!

**YES** Please include me on your mailing to receive 12 issues of 'ONLINE'.  
I would like my subscription to commence with the ..... issue.  
(insert month)

NAME .....  
ADDRESS .....  
POSTCODE .....  
I enclose (cheque/money order) for \$25.00 which includes postage and packing. OR Please charge my Bankcard  
No. ....  
Expiry Date .....  
Signature .....

you're sure to be informed of all the latest microbee developments, new products and computing information.

If paying by cheque or money order please make payable to **Applied Technology Pty. Ltd.** Post completed form to  
**Applied Technology P/L.,**  
P.O. Box 41,  
West Gosford, N.S.W. 2250.

# microbee<sup>®</sup>

## Australia's home grown computer

